
PyInstaller Documentation

Release 3.3.dev0+g8477437

David Cortesi

2017-06-17

| | | |
|----------|---|----------|
| 1 | What's New This Release | 3 |
| 1.1 | Requirements | 3 |
| 1.2 | License | 4 |
| 1.3 | How To Contribute | 4 |
| 1.4 | How to Install <i>PyInstaller</i> | 4 |
| 1.5 | What <i>PyInstaller</i> Does and How It Does It | 6 |
| 1.6 | Using PyInstaller | 9 |
| 1.7 | Run-time Information | 17 |
| 1.8 | Using Spec Files | 19 |
| 1.9 | When Things Go Wrong | 26 |
| 1.10 | Advanced Topics | 30 |
| 1.11 | Understanding PyInstaller Hooks | 37 |
| 1.12 | Building the Bootloader | 44 |
| 1.13 | Building Linux Standard Base (LSB) compliant binaries | 45 |
| 1.14 | Changelog for PyInstaller | 46 |
| 1.15 | Credits | 57 |
| 1.16 | Man Pages | 62 |
| 1.17 | Indices and tables | 68 |

Version PyInstaller 3.3.dev0+g8477437

Homepage <http://www.pyinstaller.org>

Contact pyinstaller@googlegroups.com

Authors David Cortesi, based on structure by Giovanni Bajo & William Caban, based on Gordon McMillan's manual

Copyright This document has been placed in the public domain.

PyInstaller bundles a Python application and all its dependencies into a single package. The user can run the packaged app without installing a Python interpreter or any modules. *PyInstaller* supports Python 2.7 and Python 3.3+, and correctly bundles the major Python packages such as numpy, PyQt, Django, wxPython, and others.

PyInstaller is tested against Windows, Mac OS X, and Linux. However, it is not a cross-compiler: to make a Windows app you run *PyInstaller* in Windows; to make a Linux app you run it in Linux, etc. *PyInstaller* has been used successfully with AIX, Solaris, and FreeBSD, but is not tested against them.

What's New This Release

Release 3.0 is a major rewrite that adds Python 3 support, better code quality through use of automated testing, and resolutions for many old issues.

Functional changes include removal of support for Python prior to 2.7, an easier way to include data files in the bundle (*Adding Files to the Bundle*), and changes to the “hook” API (*Understanding PyInstaller Hooks*).

Contents:

Requirements

Windows

PyInstaller runs in Windows XP or newer. It can create graphical windowed apps (apps that do not need a command window).

PyInstaller requires two Python modules in a Windows system. It requires either the [PyWin32](#) or [pypiwin32](#) Python extension for Windows. If you install *PyInstaller* using `pip`, and `PyWin32` is not already installed, `pypiwin32` is automatically installed. *PyInstaller* also requires the [pefile](#) package.

The `pip-Win` package is recommended, but not required.

Mac OS X

PyInstaller runs in Mac OS X 10.7 (Lion) or newer. It can build graphical windowed apps (apps that do not use a terminal window). *PyInstaller* builds apps that are compatible with the Mac OS X release in which you run it, and following releases. It can build 32-bit binaries in Mac OS X releases that support them.

Linux

PyInstaller requires the `ldd` terminal application to discover the shared libraries required by each program or shared library. It is typically found in the distribution-package `glibc` or `libc-bin`.

It also requires the `objdump` terminal application to extract information from object files and the `objcopy` terminal application to append data to the bootloader. These are typically found in the distribution-package `binutils`.

AIX, Solaris, and FreeBSD

Users have reported success running *PyInstaller* on these platforms, but it is not tested on them. The `ldd` and `objdump` commands are needed.

Each bundled app contains a copy of a *bootloader*, a program that sets up the application and starts it (see *The Bootstrap Process in Detail*).

When you install *PyInstaller* using `pip`, the setup will attempt to build a bootloader for this platform. If that succeeds, the installation continues and *PyInstaller* is ready to use.

If the `pip` setup fails to build a bootloader, or if you do not use `pip` to install, you must compile a bootloader manually. The process is described under *Building the Bootloader*.

License

PyInstaller is distributed under the [GPL License](#) but with an exception that allows you to use it to build commercial products:

1. You may use PyInstaller to bundle commercial applications out of your source code.
2. The executable bundles generated by PyInstaller from your source code can be shipped with whatever license you want.
3. You may modify PyInstaller for your own needs but changes to the PyInstaller source code fall under the terms of the GPL license. That is, if you distribute your modifications you must distribute them under GPL terms.

For updated information or clarification see our [FAQ](#) at the [PyInstaller](#) home page.

How To Contribute

PyInstaller is an open-source project that is created and maintained by volunteers. At [Pyinstaller.org](#) you find links to the mailing list, IRC channel, and Git repository, and the important [How to Contribute](#) link. Contributions to code and documentation are welcome, as well as tested hooks for installing other packages.

How to Install *PyInstaller*

PyInstaller is a normal Python package. You can download the archive from [PyPi](#), but it is easier to install using `pip` where it is available, for example:

```
pip install pyinstaller
```

or upgrade to a newer version:

```
pip install --upgrade pyinstaller
```

To install the current development version use:


```
pip install https://github.com/pyinstaller/pyinstaller/tarball/develop
```

Installing in Windows

For Windows, `PyWin32` or the more recent `pywin32`, is a prerequisite. The latter is installed automatically when you install *PyInstaller* using `pip` or `easy_install`. If necessary, follow the `pywin32` link to install it manually.

It is particularly easy to use `pip-Win` to install *PyInstaller* along with the correct version of `PyWin32`. `pip-Win` also provides `virtualenv`, which makes it simple to maintain multiple different Python interpreters and install packages such as *PyInstaller* in each of them. (For more on the uses of `virtualenv`, see *Supporting Multiple Platforms* below.)

When `pip-Win` is working, enter this command in its Command field and click Run:

```
venv -c -i pyi-env-name
```

This creates a new virtual environment rooted at `C:\Python\pyi-env-name` and makes it the current environment. A new command shell window opens in which you can run commands within this environment. Enter the command

```
pip install PyInstaller
```

Once it is installed, to use *PyInstaller*,

- Start `pip-Win`
- In the Command field enter `venv pyi-env-name`
- Click Run

Then you have a command shell window in which commands such as *pyinstaller* execute in that Python environment.

Installing in Mac OS X

PyInstaller works with the default Python 2.7 provided with current Mac OS X installations. However, if you plan to use a later version of Python, or if you use any of the major packages such as `PyQt`, `Numpy`, `Matplotlib`, `Scipy`, and the like, we strongly recommend that you install these using either `MacPorts` or `Homebrew`.

PyInstaller users report fewer problems when they use a package manager than when they attempt to install major packages individually.

Installing from the archive

If `pip` is not available, download the compressed archive from `PyPI`. If you are asked to test a problem using the latest development code, download the compressed archive from the `develop` branch of `PyInstaller Downloads` page.

Expand the archive. Inside is a script named `setup.py`. Execute `python setup.py install` with administrator privilege to install or upgrade *PyInstaller*.

For platforms other than Windows, Linux and Mac OS, you must first build a bootloader program for your platform: see *Building the Bootloader*. After the bootloader has been created, use `python setup.py install` with administrator privileges to complete the installation.

Verifying the installation

On all platforms, the command `pyinstaller` should now exist on the execution path. To verify this, enter the command

```
pyinstaller --version
```

The result should resemble `3.n` for a released version, and `3.n.dev0-xxxxxx` for a development branch.

If the command is not found, make sure the execution path includes the proper directory:

- Windows: `C:\PythonXY\Scripts` where `XY` stands for the major and minor Python version number, for example `C:\Python34\Scripts` for Python 3.4)
- Linux: `/usr/bin/`
- OS X (using the default Apple-supplied Python) `/usr/bin`
- OS X (using Python installed by homebrew) `/usr/local/bin`
- OS X (using Python installed by macports) `/opt/local/bin`

To display the current path in Windows the command is `echo %path%` and in other systems, `echo $PATH`.

Installed commands

The complete installation places these commands on the execution path:

- `pyinstaller` is the main command to build a bundled application. See *Using PyInstaller*.
- `pyi-makespec` is used to create a spec file. See *Using Spec Files*.
- `pyi-archive_viewer` is used to inspect a bundled application. See *Inspecting Archives*.
- `pyi-bindepend` is used to display dependencies of an executable. See *Inspecting Executables*.
- `pyi-grab_version` is used to extract a version resource from a Windows executable. See *Capturing Windows Version Data*.

If you do not perform a complete installation (installing via `pip` or executing `setup.py`), these commands will not be installed as commands. However, you can still execute all the functions documented below by running Python scripts found in the distribution folder. The equivalent of the `pyinstaller` command is `pyinstaller-folder/pyinstaller.py`. The other commands are found in `pyinstaller-folder/cliutils/` with meaningful names (`makespec.py`, etc.)

What *PyInstaller* Does and How It Does It

This section covers the basic ideas of *PyInstaller*. These ideas apply to all platforms. Options and special cases are covered below, under *Using PyInstaller*.

PyInstaller reads a Python script written by you. It analyzes your code to discover every other module and library your script needs in order to execute. Then it collects copies of all those files – including the active Python interpreter! – and puts them with your script in a single folder, or optionally in a single executable file.

For the great majority of programs, this can be done with one short command,

```
pyinstaller myscript.py
```

or with a few added options, for example a windowed application as a single-file executable,

```
pyinstaller --onefile --windowed myscript.py
```

You distribute the bundle as a folder or file to other people, and they can execute your program. To your users, the app is self-contained. They do not need to install any particular version of Python or any modules. They do not need to have Python installed at all.

Note: The output of *PyInstaller* is specific to the active operating system and the active version of Python. This means that to prepare a distribution for:

- a different OS
- a different version of Python
- a 32-bit or 64-bit OS

you run *PyInstaller* on that OS, under that version of Python. The Python interpreter that executes *PyInstaller* is part of the bundle, and it is specific to the OS and the word size.

Analysis: Finding the Files Your Program Needs

What other modules and libraries does your script need in order to run? (These are sometimes called its “dependencies”.)

To find out, *PyInstaller* finds all the `import` statements in your script. It finds the imported modules and looks in them for `import` statements, and so on recursively, until it has a complete list of modules your script may use.

PyInstaller understands the “egg” distribution format often used for Python packages. If your script imports a module from an “egg”, *PyInstaller* adds the egg and its dependencies to the set of needed files.

PyInstaller also knows about many major Python packages, including the GUI packages `Qt` (imported via `PyQt` or `PySide`), `WxPython`, `TkInter`, `Django`, and other major packages. For a complete list, see [Supported Packages](#).

Some Python scripts import modules in ways that *PyInstaller* cannot detect: for example, by using the `__import__()` function with variable data, or manipulating the `sys.path` value at run time. If your script requires files that *PyInstaller* does not know about, you must help it:

- You can give additional files on the `pyinstaller` command line.
- You can give additional import paths on the command line.
- You can edit the `myscript.spec` file that *PyInstaller* writes the first time you run it for your script. In the spec file you can tell *PyInstaller* about code modules that are unique to your script.
- You can write “hook” files that inform *PyInstaller* of hidden imports. If you create a “hook” for a package that other users might also use, you can contribute your hook file to *PyInstaller*.

If your program depends on access to certain data files, you can tell *PyInstaller* to include them in the bundle as well. You do this by modifying the spec file, an advanced topic that is covered under [Using Spec Files](#).

In order to locate included files at run time, your program needs to be able to learn its path at run time in a way that works regardless of whether or not it is running from a bundle. This is covered under [Run-time Information](#).

PyInstaller does *not* include libraries that should exist in any installation of this OS. For example in Linux, it does not bundle any file from `/lib` or `/usr/lib`, assuming these will be found in every system.

Bundling to One Folder

When you apply *PyInstaller* to `myscript.py` the default result is a single folder named `myscript`. This folder contains all your script’s dependencies, and an executable file also named `myscript` (`myscript.exe` in Windows).

You compress the folder to `myscript.zip` and transmit it to your users. They install the program simply by unzipping it. A user runs your app by opening the folder and launching the `myscript` executable inside it.

It is easy to debug problems that occur when building the app when you use one-folder mode. You can see exactly what files *PyInstaller* collected into the folder.

Another advantage of a one-folder bundle is that when you change your code, as long as it imports *exactly the same set of dependencies*, you could send out only the updated `mymscript` executable. That is typically much smaller than the entire folder. (If you change the script so that it imports more or different dependencies, or if the dependencies are upgraded, you must redistribute the whole bundle.)

A small disadvantage of the one-folder format is that the one folder contains a large number of files. Your user must find the `mymscript` executable in a long list of names or among a big array of icons. Also your user can create a problem by accidentally dragging files out of the folder.

How the One-Folder Program Works

A bundled program always starts execution in the *PyInstaller* bootloader. This is the heart of the `mymscript` executable in the folder.

The *PyInstaller* bootloader is a binary executable program for the active platform (Windows, Linux, Mac OS X, etc.). When the user launches your program, it is the bootloader that runs. The bootloader creates a temporary Python environment such that the Python interpreter will find all imported modules and libraries in the `mymscript` folder.

The bootloader starts a copy of the Python interpreter to execute your script. Everything follows normally from there, provided that all the necessary support files were included.

(This is an overview. For more detail, see *The Bootstrap Process in Detail* below.)

Bundling to One File

PyInstaller can bundle your script and all its dependencies into a single executable named `mymscript` (`mymscript.exe` in Windows).

The advantage is that your users get something they understand, a single executable to launch. A disadvantage is that any related files such as a README must be distributed separately. Also, the single executable is a little slower to start up than the one-folder bundle.

Before you attempt to bundle to one file, make sure your app works correctly when bundled to one folder. It is *much* easier to diagnose problems in one-folder mode.

How the One-File Program Works

The bootloader is the heart of the one-file bundle also. When started it creates a temporary folder in the appropriate temp-folder location for this OS. The folder is named `_MEIxxxxxx`, where `xxxxxx` is a random number.

The one executable file contains an embedded archive of all the Python modules used by your script, as well as compressed copies of any non-Python support files (e.g. `.so` files). The bootloader uncompresses the support files and writes copies into the temporary folder. This can take a little time. That is why a one-file app is a little slower to start than a one-folder app.

After creating the temporary folder, the bootloader proceeds exactly as for the one-folder bundle, in the context of the temporary folder. When the bundled code terminates, the bootloader deletes the temporary folder.

(In Linux and related systems, it is possible to mount the `/tmp` folder with a “no-execution” option. That option is not compatible with a *PyInstaller* one-file bundle. It needs to execute code out of `/tmp`.)

Because the program makes a temporary folder with a unique name, you can run multiple copies of the app; they won't interfere with each other. However, running multiple copies is expensive in disk space because nothing is shared.

The `_MEIxxxxxx` folder is not removed if the program crashes or is killed (kill -9 on Unix, killed by the Task Manager on Windows, “Force Quit” on Mac OS). Thus if your app crashes frequently, your users will lose disk space to multiple `_MEIxxxxxx` temporary folders.

Note: Do *not* give administrator privileges to a one-file executable (setuid root in Unix/Linux, or the “Run this program as an administrator” property in Windows 7). There is an unlikely but not impossible way in which a malicious attacker could corrupt one of the shared libraries in the temp folder while the bootloader is preparing it. Distribute a privileged program in one-folder mode instead.

Note: Applications that use `os.setuid()` may encounter permissions errors. The temporary folder where the bundled app runs may not be readable after `setuid` is called. If your script needs to call `setuid`, it may be better to use one-folder mode so as to have more control over the permissions on its files.

Using a Console Window

By default the bootloader creates a command-line console (a terminal window in Linux and Mac OS, a command window in Windows). It gives this window to the Python interpreter for its standard input and output. Your script’s use of `print` and `input()` are directed here. Error messages from Python and default logging output also appear in the console window.

An option for Windows and Mac OS is to tell *PyInstaller* to not provide a console window. The bootloader starts Python with no target for standard output or input. Do this when your script has a graphical interface for user input and can properly report its own diagnostics.

Hiding the Source Code

The bundled app does not include any source code. However, *PyInstaller* bundles compiled Python scripts (`.pyc` files). These could in principle be decompiled to reveal the logic of your code.

If you want to hide your source code more thoroughly, one possible option is to compile some of your modules with *Cython*. Using *Cython* you can convert Python modules into C and compile the C to machine language. *PyInstaller* can follow import statements that refer to *Cython* C object modules and bundle them.

Additionally, Python bytecode can be obfuscated with AES256 by specifying an encryption key on *PyInstaller*’s command line. Please note that it is still very easy to extract the key and get back the original bytecode, but it should prevent most forms of “casual” tampering.

Using PyInstaller

The syntax of the `pyinstaller` command is:

```
pyinstaller [options] script [script ...] | specfile
```

In the most simple case, set the current directory to the location of your program `myscript.py` and execute:

```
pyinstaller myscript.py
```

PyInstaller analyzes `myscript.py` and:

- Writes `myscript.spec` in the same folder as the script.

- Creates a folder `build` in the same folder as the script if it does not exist.
- Writes some log files and working files in the `build` folder.
- Creates a folder `dist` in the same folder as the script if it does not exist.
- Writes the `myscript` executable folder in the `dist` folder.

In the `dist` folder you find the bundled app you distribute to your users.

Normally you name one script on the command line. If you name more, all are analyzed and included in the output. However, the first script named supplies the name for the spec file and for the executable folder or file. Its code is the first to execute at run-time.

For certain uses you may edit the contents of `myscript.spec` (described under *Using Spec Files*). After you do this, you name the spec file to *PyInstaller* instead of the script:

```
pyinstaller myscript.spec
```

You may give a path to the script or spec file, for example

```
pyinstaller options... ~/myproject/source/myscript.py
```

or, on Windows,

```
pyinstaller "C:\Documents and Settings\project\myscript.spec"
```

Options

General Options

- | | |
|----------------------------|---|
| -h, --help | show this help message and exit |
| -v, --version | Show program version info and exit. |
| --distpath DIR | Where to put the bundled app (default: <code>./dist</code>) |
| --workpath WORKPATH | Where to put all the temporary work files, <code>.log</code> , <code>.pyz</code> and etc. (default: <code>./build</code>) |
| -y, --noconfirm | Replace output directory (default: <code>SPECPATH/dist/SPECNAME</code>) without asking for confirmation |
| --upx-dir UPX_DIR | Path to UPX utility (default: search the execution path) |
| -a, --ascii | Do not include unicode encoding support (default: included if available) |
| --clean | Clean PyInstaller cache and remove temporary files before building. |
| --log-level LEVEL | Amount of detail in build-time console messages. <code>LEVEL</code> may be one of <code>TRACE</code> , <code>DEBUG</code> , <code>INFO</code> , <code>WARN</code> , <code>ERROR</code> , <code>CRITICAL</code> (default: <code>INFO</code>). |

What to generate

- | | |
|-----------------------------|--|
| -D, --onedir | Create a one-folder bundle containing an executable (default) |
| -F, --onefile | Create a one-file bundled executable. |
| --specpath DIR | Folder to store the generated spec file (default: current directory) |
| -n NAME, --name NAME | Name to assign to the bundled app and spec file (default: first script's basename) |

What to bundle, where to search

- add-data <SRC;DEST or SRC:DEST>** Additional non-binary files or folders to be added to the executable. The path separator is platform specific, `os.pathsep` (which is `;` on Windows and `:` on most unix systems) is used. This option can be used multiple times.
- add-binary <SRC;DEST or SRC:DEST>** Additional binary files to be added to the executable. See the `--add-data` option for more details. This option can be used multiple times.
- p DIR, --paths DIR** A path to search for imports (like using `PYTHONPATH`). Multiple paths are allowed, separated by `'.'`, or use this option multiple times
- hidden-import MODULENAME, --hiddenimport MODULENAME** Name an import not visible in the code of the script(s). This option can be used multiple times.
- additional-hooks-dir HOOKSPATH** An additional path to search for hooks. This option can be used multiple times.
- runtime-hook RUNTIME_HOOKS** Path to a custom runtime hook file. A runtime hook is code that is bundled with the executable and is executed before any other code or module to set up special features of the runtime environment. This option can be used multiple times.
- exclude-module EXCLUDES** Optional module or package (the Python name, not the path name) that will be ignored (as though it was not found). This option can be used multiple times.
- key KEY** The key used to encrypt Python bytecode.

How to generate

- d, --debug** Tell the bootloader to issue progress messages while initializing and starting the bundled app. Used to diagnose problems with missing imports.
- s, --strip** Apply a symbol-table strip to the executable and shared libs (not recommended for Windows)
- noupX** Do not use UPX even if it is available (works differently between Windows and *nix)

Windows and Mac OS X specific options

- c, --console, --nowindowed** Open a console window for standard i/o (default)
- w, --windowed, --noconsole** Windows and Mac OS X: do not provide a console window for standard i/o. On Mac OS X this also triggers building an OS X `.app` bundle. This option is ignored in *NIX systems.
- i <FILE.ico or FILE.exe,ID or FILE.icns>, --icon <FILE.ico or FILE.exe,ID or FILE.icns>**
 FILE.ico: apply that icon to a Windows executable. FILE.exe,ID, extract the icon with ID from an exe. FILE.icns: apply the icon to the `.app` bundle on Mac OS X

Windows specific options

- version-file FILE** add a version resource from FILE to the exe
- m <FILE or XML>, --manifest <FILE or XML>** add manifest FILE or XML to the exe
- r RESOURCE, --resource RESOURCE** Add or update a resource to a Windows executable. The RESOURCE is one to four items, FILE[,TYPE[,NAME[,LANGUAGE]]]. FILE can be a data file or an exe/dll. For data files, at least TYPE and NAME must be specified. LANGUAGE defaults to 0 or may be specified as wildcard * to update all resources of the given TYPE and NAME. For exe/dll files, all resources from FILE will be added/updated to the final executable if TYPE, NAME and LANGUAGE are omitted or specified as wildcard *. This option can be used multiple times.
- uac-admin** Using this option creates a Manifest which will request elevation upon application restart.
- uac-uiaccess** Using this option allows an elevated application to work with Remote Desktop.

Windows Side-by-side Assembly searching options (advanced)

- win-private-assemblies** Any Shared Assemblies bundled into the application will be changed into Private Assemblies. This means the exact versions of these assemblies will always be used, and any newer versions installed on user machines at the system level will be ignored.
- win-no-prefer-redirects** While searching for Shared or Private Assemblies to bundle into the application, PyInstaller will prefer not to follow policies that redirect to newer versions, and will try to bundle the exact versions of the assembly.

Mac OS X specific options

- osx-bundle-identifier BUNDLE_IDENTIFIER** Mac OS X .app bundle identifier is used as the default unique program name for code signing purposes. The usual form is a hierarchical name in reverse DNS notation. For example: com.mycompany.department.appname (default: first script's basename)

Shortening the Command

Because of its numerous options, a full `pyinstaller` command can become very long. You will run the same command again and again as you develop your script. You can put the command in a shell script or batch file, using line continuations to make it readable. For example, in Linux:

```
pyinstaller --noconfirm --log-level=WARN \  
  --onefile --nowindow \  
  --add-data="README:." \  
  --add-data="image1.png:img" \  
  --add-binary="libfoo.so:lib" \  
  --hidden-import=secret1 \  
  --hidden-import=secret2 \  
  --upx-dir=/usr/local/share/ \  
  myscript.spec
```

Or in Windows, use the little-known BAT file line continuation:


```

pyinstaller --noconfirm --log-level=WARN ^
  --onefile --nowindow ^
  --add-data="README;." ^
  --add-data="image1.png;img" ^
  --add-binary="libfoo.so;lib" ^
  --hidden-import=secret1 ^
  --hidden-import=secret2 ^
  --icon=..\MLNMF.LCN.ICO ^
  myscript.spec

```

Using UPX

UPX is a free utility available for most operating systems. UPX compresses executable files and libraries, making them smaller, sometimes much smaller. UPX is available for most operating systems and can compress a large number of executable file formats. See the [UPX home page](#) for downloads, and for the list of supported executable formats.

A compressed executable program is wrapped in UPX startup code that dynamically decompresses the program when the program is launched. After it has been decompressed, the program runs normally. In the case of a *PyInstaller* one-file executable that has been UPX-compressed, the full execution sequence is:

- The compressed program start up in the UPX decompressor code.
- After decompression, the program executes the *PyInstaller* bootloader, which creates a temporary environment for Python.
- The Python interpreter executes your script.

PyInstaller looks for UPX on the execution path or the path specified with the `--upx-dir` option. If UPX exists, *PyInstaller* applies it to the final executable, unless the `--noupx` option was given. UPX has been used with *PyInstaller* output often, usually with no problems.

Encrypting Python Bytecode

To encrypt the Python bytecode modules stored in the bundle, pass the `--key=key-string` argument on the command line.

For this to work, you must have the [PyCrypto](#) module installed. The *key-string* is a string of 16 characters which is used to encrypt each file of Python byte-code before it is stored in the archive inside the executable file.

Supporting Multiple Platforms

If you distribute your application for only one combination of OS and Python, just install *PyInstaller* like any other package and use it in your normal development setup.

Supporting Multiple Python Environments

When you need to bundle your application within one OS but for different versions of Python and support libraries – for example, a Python 3 version and a Python 2.7 version; or a supported version that uses Qt4 and a development version that uses Qt5 – we recommend you use [virtualenv](#). With *virtualenv* you can maintain different combinations of Python and installed packages, and switch from one combination to another easily. (If you work only with Python 3.4 and later, the built-in script [pyvenv](#) does the same job.)

- Use *virtualenv* to create as many different development environments as you need, each with its unique combination of Python and installed packages.

- Install *PyInstaller* in each environment.
- Use *PyInstaller* to build your application in each environment.

Note that when using virtualenv, the path to the *PyInstaller* commands is:

- Windows: ENV_ROOT\Scripts
- Others: ENV_ROOT/bin

Under Windows, the [pip-Win](#) package installs virtualenv and makes it especially easy to set up different environments and switch between them. Under Linux and Mac OS, you switch environments at the command line.

Supporting Multiple Operating Systems

If you need to distribute your application for more than one OS, for example both Windows and Mac OS X, you must install *PyInstaller* on each platform and bundle your app separately on each.

You can do this from a single machine using virtualization. The free [virtualBox](#) or the paid [VMWare](#) and [Parallels](#) allow you to run another complete operating system as a “guest”. You set up a virtual machine for each “guest” OS. In it you install Python, the support packages your application needs, and *PyInstaller*.

The [Dropbox](#) system is useful with virtual machines. Install a Dropbox client in each virtual machine, all linked to your Dropbox account. Keep a single copy of your script(s) in a Dropbox folder. Then on any virtual machine you can run *PyInstaller* thus:

```
cd ~/Dropbox/project_folder/src # Linux, Mac -- Windows similar
rm *.pyc # get rid of modules compiled by another Python
pyinstaller --workpath=path-to-local-temp-folder \
            --distpath=path-to-local-dist-folder \
            ...other options as required... \
            ./myscript.py
```

PyInstaller reads scripts from the common Dropbox folder, but writes its work files and the bundled app in folders that are local to the virtual machine.

If you share the same home directory on multiple platforms, for example Linux and OS X, you will need to set the `PYINSTALLER_CONFIG_DIR` environment variable to different values on each platform otherwise *PyInstaller* may cache files for one platform and use them on the other platform, as by default it uses a subdirectory of your home directory as its cache location.

It is said to be possible to cross-develop for Windows under Linux using the free [Wine](#) environment. Further details are needed, see [How to Contribute](#).

Making Linux Apps Forward-Compatible

Under Linux, *PyInstaller* does not bundle `libc` (the C standard library, usually `glibc`, the Gnu version) with the app. Instead, the app expects to link dynamically to the `libc` from the local OS where it runs. The interface between any app and `libc` is forward compatible to newer releases, but it is not backward compatible to older releases.

For this reason, if you bundle your app on the current version of Linux, it may fail to execute (typically with a runtime dynamic link error) if it is executed on an older version of Linux.

The solution is to always build your app on the *oldest* version of Linux you mean to support. It should continue to work with the `libc` found on newer versions.

The Linux standard libraries such as `glibc` are distributed in 64-bit and 32-bit versions, and these are not compatible. As a result you cannot bundle your app on a 32-bit system and run it on a 64-bit installation, nor vice-versa. You must make a unique version of the app for each word-length supported.

Capturing Windows Version Data

A Windows app may require a Version resource file. A Version resource contains a group of data structures, some containing binary integers and some containing strings, that describe the properties of the executable. For details see the Microsoft [Version Information Structures](#) page.

Version resources are complex and some elements are optional, others required. When you view the version tab of a Properties dialog, there's no simple relationship between the data displayed and the structure of the resource. For this reason *PyInstaller* includes the `pyi-grab_version` command. It is invoked with the full path name of any Windows executable that has a Version resource:

```
pyi-grab_version executable_with_version_resource
```

The command writes text that represents a Version resource in readable form to standard output. You can copy it from the console window or redirect it to a file. Then you can edit the version information to adapt it to your program. Using `pyi-grab_version` you can find an executable that displays the kind of information you want, copy its resource data, and modify it to suit your package.

The version text file is encoded UTF-8 and may contain non-ASCII characters. (Unicode characters are allowed in Version resource string fields.) Be sure to edit and save the text file in UTF-8 unless you are certain it contains only ASCII string values.

Your edited version text file can be given with the `--version-file=` option to `pyinstaller` or `pyi-makespec`. The text data is converted to a Version resource and installed in the bundled app.

In a Version resource there are two 64-bit binary values, `FileVersion` and `ProductVersion`. In the version text file these are given as four-element tuples, for example:

```
filevers=(2, 0, 4, 0),
prodvers=(2, 0, 4, 0),
```

The elements of each tuple represent 16-bit values from most-significant to least-significant. For example the value `(2, 0, 4, 0)` resolves to `00020000000040000` in hex.

You can also install a Version resource from a text file after the bundled app has been created, using the `pyi-set_version` command:

```
pyi-set_version version_text_file executable_file
```

The `pyi-set_version` utility reads a version text file as written by `pyi-grab_version`, converts it to a Version resource, and installs that resource in the `executable_file` specified.

For advanced uses, examine a version text file as written by `pyi-grab_version`. You find it is Python code that creates a `VSVersionInfo` object. The class definition for `VSVersionInfo` is found in `utils/win32/versioninfo.py` in the *PyInstaller* distribution folder. You can write a program that imports `versioninfo`. In that program you can `eval` the contents of a version info text file to produce a `VSVersionInfo` object. You can use the `.toRaw()` method of that object to produce a Version resource in binary form. Or you can apply the `unicode()` function to the object to reproduce the version text file.

Building Mac OS X App Bundles

If you specify only `--onefile` under Mac OS X, the output in `dist` is a UNIX executable `myscript`. It can be executed from a Terminal command line. Standard input and output work as normal through the Terminal window.

If you also specify `--windowed`, the `dist` folder contains two outputs: the UNIX executable `myscript` and also an OS X application named `myscript.app`.

As you probably know, an application is a special type of folder. The one built by *PyInstaller* contains a folder always named `Contents`. It contains:

- A folder `Frameworks` which is empty.
- A folder `MacOS` that contains a copy of the same `myscript` UNIX executable.
- A folder `Resources` that contains an icon file.
- A file `Info.plist` that describes the app.

PyInstaller builds minimal versions of these elements.

Use the `osx-bundle-identifier=` argument to add a bundle identifier. This becomes the `CFBundleIdentifier` used in code-signing (see the [PyInstaller code signing recipe](#) and for more detail, the [Apple code signing overview](#) technical note).

Use the `icon=` argument to specify a custom icon for the application. (If you do not specify an icon file, *PyInstaller* supplies a file `icon-windowed.icns` with the *PyInstaller* logo.)

You can add items to the `Info.plist` by editing the spec file; see [Spec File Options for a Mac OS X Bundle](#) below.

Making Mac OS X apps Forward-Compatible

In Mac OS X, components from one version of the OS are usually compatible with later versions, but they may not work with earlier versions.

The only way to be certain your app supports an older version of Mac OS X is to run *PyInstaller* in the oldest version of the OS you need to support.

For example, to be sure of compatibility with “Snow Leopard” (10.6) and later versions, you should execute *PyInstaller* in that environment. You would create a copy of Mac OS X 10.6, typically in a virtual machine. In it, install the desired level of Python (the default Python in Snow Leopard was 2.6, which *PyInstaller* no longer supports), and install *PyInstaller*, your source, and all its dependencies. Then build your app in that environment. It should be compatible with later versions of Mac OS X.

Building 32-bit Apps in Mac OS X

Older versions of Mac OS X supported both 32-bit and 64-bit executables. *PyInstaller* builds an app using the the word-length of the Python used to execute it. That will typically be a 64-bit version of Python, resulting in a 64-bit executable. To create a 32-bit executable, run *PyInstaller* under a 32-bit Python.

Python as installed in OS X will usually be executable in either 64- or 32-bit mode. To verify this, apply the `file` command to the Python executable:

```
$ file /usr/local/bin/python3
/usr/local/bin/python3: Mach-O universal binary with 2 architectures
/usr/local/bin/python3 (for architecture i386):      Mach-O executable i386
/usr/local/bin/python3 (for architecture x86_64):    Mach-O 64-bit executable x86_64
```

The OS chooses which architecture to run, and typically defaults to 64-bit. You can force the use of either architecture by name using the `arch` command:

```
$ /usr/local/bin/python3
Python 3.4.2 (v3.4.2:ab2c023a9432, Oct  5 2014, 20:42:22)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys; sys.maxsize
9223372036854775807

$ arch -i386 /usr/local/bin/python3
Python 3.4.2 (v3.4.2:ab2c023a9432, Oct  5 2014, 20:42:22)
```

```
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys; sys.maxsize
2147483647
```

Apple’s default `/usr/bin/python` may circumvent the `arch` specification and run 64-bit regardless. (That is not the case if you apply `arch` to a specific version such as `/usr/bin/python2.7`.) To make sure of running 32-bit in all cases, set the following environment variable:

```
VERSIONER_PYTHON_PREFER_32_BIT=yes
arch -i386 /usr/bin/python pyinstaller --clean -F -w myscript.py
```

Getting the Opened Document Names

Note: Support for OpenDocument events is broken in *PyInstaller* 3.0 owing to code changes needed in the bootloader to support current versions of Mac OS X. Do not attempt to use this feature until it has been fixed. If this feature is important to you, follow and comment on the status of [PyInstaller Issue #1309](#).

When a user double-clicks a document of a type your application supports, or when a user drags a document icon and drops it on your application’s icon, Mac OS X launches your application and provides the name(s) of the opened document(s) in the form of an OpenDocument AppleEvent. This AppleEvent is received by the bootloader before your code has started executing.

The bootloader gets the names of opened documents from the OpenDocument event and encodes them into the `argv` string before starting your code. Thus your code can query `sys.argv` to get the names of documents that should be opened at startup.

OpenDocument is the only AppleEvent the bootloader handles. If you want to handle other events, or events that are delivered after the program has launched, you must set up the appropriate handlers.

Run-time Information

Your app should run in a bundle exactly as it does when run from source. However, you may need to learn at run-time whether the app is running from source, or is “frozen” (bundled). For example, you might have data files that are normally found based on a module’s `__file__` attribute. That will not work when the code is bundled.

The *PyInstaller* bootloader adds the name `frozen` to the `sys` module. So the test for “are we bundled?” is:

```
import sys
if getattr( sys, 'frozen', False ) :
    # running in a bundle
else :
    # running live
```

When your app is running, it may need to access data files in any of three general locations:

- Files that were bundled with it (see [Adding Data Files](#)).
- Files the user has placed with the app bundle, say in the same folder.
- Files in the user’s current working directory.

The program has access to several path variables for these uses.

Using `__file__` and `sys._MEIPASS`

When your program is not frozen, the standard Python variable `__file__` is the full path to the script now executing. When a bundled app starts up, the bootloader sets the `sys.frozen` attribute and stores the absolute path to the bundle folder in `sys._MEIPASS`. For a one-folder bundle, this is the path to that folder, wherever the user may have put it. For a one-file bundle, this is the path to the `_MEIxxxxxxx` temporary folder created by the bootloader (see *How the One-File Program Works*).

Using `sys.executable` and `sys.argv[0]`

When a normal Python script runs, `sys.executable` is the path to the program that was executed, namely, the Python interpreter. In a frozen app, `sys.executable` is also the path to the program that was executed, but that is not Python; it is the bootloader in either the one-file app or the executable in the one-folder app. This gives you a reliable way to locate the frozen executable the user actually launched.

The value of `sys.argv[0]` is the name or relative path that was used in the user's command. It may be a relative path or an absolute path depending on the platform and how the app was launched.

If the user launches the app by way of a symbolic link, `sys.argv[0]` uses that symbolic name, while `sys.executable` is the actual path to the executable. Sometimes the same app is linked under different names and is expected to behave differently depending on the name that is used to launch it. For this case, you would test `os.path.basename(sys.argv[0])`

On the other hand, sometimes the user is told to store the executable in the same folder as the files it will operate on, for example a music player that should be stored in the same folder as the audio files it will play. For this case, you would use `os.path.dirname(sys.executable)`.

The following small program explores some of these possibilities. Save it as `directories.py`. Execute it as a Python script, then bundled as a one-folder app. Then bundle it as a one-file app and launch it directly and also via a symbolic link:

```
#!/usr/bin/python3
import sys, os
frozen = 'not'
if getattr(sys, 'frozen', False):
    # we are running in a bundle
    frozen = 'ever so'
    bundle_dir = sys._MEIPASS
else:
    # we are running in a normal Python environment
    bundle_dir = os.path.dirname(os.path.abspath(__file__))
print( 'we are',frozen,'frozen' )
print( 'bundle dir is', bundle_dir )
print( 'sys.argv[0] is', sys.argv[0] )
print( 'sys.executable is', sys.executable )
print( 'os.getcwd is', os.getcwd() )
```

LD_LIBRARY_PATH / LIBPATH considerations

This environment variable is used to discover libraries, it is the library search path - on Linux and *BSD `LD_LIBRARY_PATH` is used, on AIX it is `LIBPATH`.

PyInstaller saves the original value to `*_ORIG`, then modifies the search path so that the bundled libraries are found first by the bundled code.

But if your code executes a system program, you often do not want that this system program loads your bundled libraries (that are maybe not compatible with your system program) - it rather should load the correct libraries from the system locations like it usually does.

Thus you need to restore the original path before creating the subprocess with the system program.

```
env = dict(os.environ) # make a copy of the environment
lp_key = 'LD_LIBRARY_PATH' # for Linux and *BSD.
lp_orig = env.get(lp_key + '_ORIG') # pyinstaller >= 20160820 has this
if lp_orig is not None:
    env[lp_key] = lp_orig # restore the original, unmodified value
else:
    env.pop(lp_key, None) # last resort: remove the env var
p = Popen(system_cmd, ..., env=env) # create the process
```

Using Spec Files

When you execute

```
pyinstaller options..myscript.py
```

the first thing *PyInstaller* does is to build a spec (specification) file `myscript.spec`. That file is stored in the `--specpath=` directory, by default the current directory.

The spec file tells *PyInstaller* how to process your script. It encodes the script names and most of the options you give to the `pyinstaller` command. The spec file is actually executable Python code. *PyInstaller* builds the app by executing the contents of the spec file.

For many uses of *PyInstaller* you do not need to examine or modify the spec file. It is usually enough to give all the needed information (such as hidden imports) as options to the `pyinstaller` command and let it run.

There are four cases where it is useful to modify the spec file:

- When you want to bundle data files with the app.
- When you want to include run-time libraries (`.dll` or `.so` files) that *PyInstaller* does not know about from any other source.
- When you want to add Python run-time options to the executable.
- When you want to create a multiprogram bundle with merged common modules.

These uses are covered in topics below.

You create a spec file using this command:

```
pyi-makespec options name.py [other scripts ...]
```

The *options* are the same options documented above for the `pyinstaller` command. This command creates the *name.spec* file but does not go on to build the executable.

After you have created a spec file and modified it as necessary, you build the application by passing the spec file to the `pyinstaller` command:

```
pyinstaller options name.spec
```

When you create a spec file, most command options are encoded in the spec file. When you build from a spec file, those options cannot be changed. If they are given on the command line they are ignored and replaced by the options in the spec file.

Only the following command-line options have an effect when building from a spec file:

- `--upx-dir=`
- `--distpath=`
- `--workpath=`
- `--noconfirm`
- `--ascii`
- `--clean`

Spec File Operation

After *PyInstaller* creates a spec file, or opens a spec file when one is given instead of a script, the `pyinstaller` command executes the spec file as code. Your bundled application is created by the execution of the spec file. The following is an shortened example of a spec file for a minimal, one-folder app:

```
block_cipher = None
a = Analysis(['minimal.py'],
            pathex=['/Developer/PItests/minimal'],
            binaries=None,
            datas=None,
            hiddenimports=[],
            hookspath=None,
            runtime_hooks=None,
            excludes=None,
            cipher=block_cipher)
pyz = PYZ(a.pure, a.zipped_data,
          cipher=block_cipher)
exe = EXE(pyz, ...)
coll = COLLECT(...)
```

The statements in a spec file create instances of four classes, `Analysis`, `PYZ`, `EXE` and `COLLECT`.

- A new instance of class `Analysis` takes a list of script names as input. It analyzes all imports and other dependencies. The resulting object (assigned to `a`) contains lists of dependencies in class members named:
 - `scripts`: the python scripts named on the command line;
 - `pure`: pure python modules needed by the scripts;
 - `binaries`: non-python modules needed by the scripts, including names given by the `--add-binary` option;
 - `datas`: non-binary files included in the app, including names given by the `--add-data` option.
- An instance of class `PYZ` is a `.pyz` archive (described under *Inspecting Archives* below), which contains all the Python modules from `a.pure`.
- An instance of `EXE` is built from the analyzed scripts and the `PYZ` archive. This object creates the executable file.
- An instance of `COLLECT` creates the output folder from all the other parts.

In one-file mode, there is no call to `COLLECT`, and the `EXE` instance receives all of the scripts, modules and binaries.

You modify the spec file to pass additional values to `Analysis` and to `EXE`.

Adding Files to the Bundle

To add files to the bundle, you create a list that describes the files and supply it to the `Analysis` call. When you bundle to a single folder (see *Bundling to One Folder*), the added data files are copied into the folder with the executable. When you bundle to a single executable (see *Bundling to One File*), copies of added files are compressed into the executable, and expanded to the `_MEIxxxxxx` temporary folder before execution. This means that any changes a one-file executable makes to an added file will be lost when the application ends.

In either case, to find the data files at run-time, see *Run-time Information*.

Adding Data Files

You can add data files to the bundle by using the `--add-data` command option, or by adding them as a list to the spec file.

When using the spec file, provide a list that describes the files as the value of the `datas=` argument to `Analysis`. The list of data files is a list of tuples. Each tuple has two values, both of which must be strings:

- The first string specifies the file or files as they are in this system now.
- The second specifies the name of the *folder* to contain the files at run-time.

For example, to add a single README file to the top level of a one-folder app, you could modify the spec file as follows:

```
a = Analysis(...
    datas=[ ('src/README.txt', '.') ],
    ...
)
```

And the command line equivalent (see *What to bundle, where to search* for platform-specific details):

```
pyinstaller --add-data 'src/README.txt:.' myscript.py
```

You have made the `datas=` argument a one-item list. The item is a tuple in which the first string says the existing file is `src/README.txt`. That file will be looked up (relative to the location of the spec file) and copied into the top level of the bundled app.

The strings may use either `/` or `\` as the path separator character. You can specify input files using “glob” abbreviations. For example to include all the `.mp3` files from a certain folder:

```
a = Analysis(...
    datas= [ ('/mygame/sfx/*.mp3', 'sfx' ) ],
    ...
)
```

All the `.mp3` files in the folder `/mygame/sfx` will be copied into a folder named `sfx` in the bundled app.

The spec file is more readable if you create the list of added files in a separate statement:

```
added_files = [
    ('src/README.txt', '.')
    ('/mygame/sfx/*.mp3', 'sfx' ),
]
a = Analysis(...
    datas = added_files,
    ...
)
```

You can also include the entire contents of a folder:

```
added_files = [
    ( 'src/README.txt', '.' )
    ( '/mygame/data', 'data' ),
    ( '/mygame/sfx/*.mp3', 'sfx' ),
]
```

The folder `/mygame/data` will be reproduced under the name `data` in the bundle.

Using Data Files from a Module

If the data files you are adding are contained within a Python module, you can retrieve them using `pkgutils.get_data()`.

For example, suppose that part of your application is a module named `helpmod`. In the same folder as your script and its spec file you have this folder arrangement:

```
helpmod
  __init__.py
  helpmod.py
  help_data.txt
```

Because your script includes the statement `import helpmod`, *PyInstaller* will create this folder arrangement in your bundled app. However, it will only include the `.py` files. The data file `help_data.txt` will not be automatically included. To cause it to be included also, you would add a `datas` tuple to the spec file:

```
a = Analysis(...
    datas= [ ('helpmod/help_data.txt', 'helpmod' ) ],
    ...
)
```

When your script executes, you could find `help_data.txt` by using its base folder path, as described in the previous section. However, this data file is part of a module, so you can also retrieve its contents using the standard library function `pkgutil.get_data()`:

```
import pkgutil
help_bin = pkgutil.get_data( 'helpmod', 'help_data.txt' )
```

In Python 3, this returns the contents of the `help_data.txt` file as a binary string. If it is actually characters, you must decode it:

```
help_utf = help_bin.decode('UTF-8', 'ignore')
```

Adding Binary Files

You can add binary files to the bundle by using the `--add-binary` command option, or by adding them as a list to the spec file.

In the spec file, make a list of tuples that describe the files needed. Assign the list of tuples to the `binaries=` argument of `Analysis`.

Normally *PyInstaller* learns about `.so` and `.dll` libraries by analyzing the imported modules. Sometimes it is not clear that a module is imported; in that case you use a `--hidden-import=` command option. But even that might not find all dependencies.

Suppose you have a module `special_ops.so` that is written in C and uses the Python C-API. Your program imports `special_ops`, and *PyInstaller* finds and includes `special_ops.so`. But perhaps `special_ops.so` links to `libiodbc.2.dylib`. *PyInstaller* does not find this dependency. You could add it to the bundle this way:

```
a = Analysis(...
    binaries=[ ( '/usr/lib/libiodbc.2.dylib', '.' ) ],
    ...
```

Or via the command line (again, see *What to bundle, where to search* for platform-specific details):

```
pyinstaller --add-binary '/usr/lib/libiodbc.2.dylib:.' myscript.py
```

As with data files, if you have multiple binary files to add, create the list in a separate statement and pass the list by name.

Advanced Methods of Adding Files

PyInstaller supports a more advanced (and complex) way of adding files to the bundle that may be useful for special cases. See *The TOC and Tree Classes* below.

Giving Run-time Python Options

You can pass command-line options to the Python interpreter. The interpreter takes a number of command-line options but only the following are supported for a bundled app:

- `v` to write a message to stdout each time a module is initialized.
- `u` for unbuffered stdio.
- `W` and an option to change warning behavior: `W ignore` or `W once` or `W error`.

To pass one or more of these options, create a list of tuples, one for each option, and pass the list as an additional argument to the EXE call. Each tuple has three elements:

- The option as a string, for example `v` or `W ignore`.
- `None`
- The string `OPTION`

For example modify the spec file this way:

```
options = [ ('v', None, 'OPTION'), ('W ignore', None, 'OPTION') ]
a = Analysis( ...
    )
...
exe = EXE(pyz,
    a.scripts,
    options, <--- added line
    exclude_binaries=...
    )
```

Spec File Options for a Mac OS X Bundle

When you build a windowed Mac OS X app (that is, running in Mac OS X, you specify the `--onefile` `--windowed` options), the spec file contains an additional statement to create the Mac OS X application bundle, or app folder:

```
app = BUNDLE(exe,
              name='myscript.app',
              icon=None,
              bundle_identifier=None)
```

The `icon=` argument to `BUNDLE` will have the path to an icon file that you specify using the `--icon=` option. The `bundle_identifier` will have the value you specify with the `--osx-bundle-identifier=` option.

An `Info.plist` file is an important part of a Mac OS X app bundle. (See the [Apple bundle overview](#) for a discussion of the contents of `Info.plist`.)

PyInstaller creates a minimal `Info.plist`. You can add or overwrite entries in the plist by passing an `info_plist=` parameter to the `BUNDLE` call. The value of this argument is a Python dict. Each key and value in the dict becomes a key and value in the `Info.plist` file. For example, when you use PyQt5, you can set `NSHighResolutionCapable` to `True` to let your app also work in retina screen:

```
app = BUNDLE(exe,
              name='myscript.app',
              icon=None,
              bundle_identifier=None,
              info_plist={
                  'NSHighResolutionCapable': 'True'
              },
              )
```

The `info_plist=` parameter only handles simple key:value pairs. It cannot handle nested XML arrays. For example, if you want to modify `Info.plist` to tell Mac OS X what filetypes your app supports, you must add a `CFBundleDocumentTypes` entry to `Info.plist` (see [Apple document types](#)). The value of that keyword is a list of dicts, each containing up to five key:value pairs.

To add such a value to your app's `Info.plist` you must edit the plist file separately after *PyInstaller* has created the app. However, when you re-run *PyInstaller*, your changes will be wiped out. One solution is to prepare a complete `Info.plist` file and copy it into the app after creating it.

Begin by building and testing the windowed app. When it works, copy the `Info.plist` prepared by *PyInstaller*. This includes the `CFBundleExecutable` value as well as the icon path and bundle identifier if you supplied them. Edit the `Info.plist` as necessary to add more items and save it separately.

From that point on, to rebuild the app call *PyInstaller* in a shell script, and follow it with a statement such as:

```
cp -f Info.plist dist/myscript.app/Contents/Info.plist
```

Multipackage Bundles

Note: This feature is broken in the *PyInstaller* 3.0 release. Do not attempt building multipackage bundles until the feature is fixed. If this feature is important to you, follow and comment on [PyInstaller Issue #1527](#).

Some products are made of several different apps, each of which might depend on a common set of third-party libraries, or share code in other ways. When packaging such a product it would be a pity to treat each app in isolation, bundling it with all its dependencies, because that means storing duplicate copies of code and libraries.

You can use the multipackage feature to bundle a set of executable apps so that they share single copies of libraries. You can do this with either one-file or one-folder apps. Each dependency (a DLL, for example) is packaged only once, in one of the apps. Any other apps in the set that depend on that DLL have an “external reference” to it, telling them to extract that dependency from the executable file of the app that contains it.

This saves disk space because each dependency is stored only once. However, to follow an external reference takes extra time when an app is starting up. All but one of the apps in the set will have slightly slower launch times.

The external references between binaries include hard-coded paths to the output directory, and cannot be rearranged. If you use one-folder mode, you must install all the application folders within a single parent directory. If you use one-file mode, you must place all the related applications in the same directory when you install the application.

To build such a set of apps you must code a custom spec file that contains a call to the `MERGE` function. This function takes a list of analyzed scripts, finds their common dependencies, and modifies the analyses to minimize the storage cost.

The order of the analysis objects in the argument list matters. The `MERGE` function packages each dependency into the first script from left to right that needs that dependency. A script that comes later in the list and needs the same file will have an external reference to the prior script in the list. You might sequence the scripts to place the most-used scripts first in the list.

A custom spec file for a multipackage bundle contains one call to the `MERGE` function:

```
MERGE(*args)
```

`MERGE` is used after the analysis phase and before `EXE` and `COLLECT`. Its variable-length list of arguments consists of a list of tuples, each tuple having three elements:

- The first element is an `Analysis` object, an instance of class `Analysis`, as applied to one of the apps.
- The second element is the script name of the analyzed app (without the `.py` extension).
- The third element is the name for the executable (usually the same as the script).

`MERGE` examines the `Analysis` objects to learn the dependencies of each script. It modifies these objects to avoid duplication of libraries and modules. As a result the packages generated will be connected.

Example `MERGE` spec file

One way to construct a spec file for a multipackage bundle is to first build a spec file for each app in the package. Suppose you have a product that comprises three apps named (because we have no imagination) `foo`, `bar` and `zap`:

```
pyi-makespec options as appropriate... foo.py
pyi-makespec options as appropriate... bar.py
pyi-makespec options as appropriate... zap.py
```

Check for warnings and test each of the apps individually. Deal with any hidden imports and other problems. When all three work correctly, combine the statements from the three files `foo.spec`, `bar.spec` and `zap.spec` as follows.

First copy the `Analysis` statements from each, changing them to give each `Analysis` object a unique name:

```
foo_a = Analysis(['foo.py'],
                 pathex=['/the/path/to/foo'],
                 hiddenimports=[],
                 hookspath=None)

bar_a = Analysis(['bar.py'], etc., etc...)

zap_a = Analysis(['zap.py'], etc., etc...)
```

Now call the `MERGE` method to process the three `Analysis` objects:

```
MERGE( (foo_a, 'foo', 'foo'), (bar_a, 'bar', 'bar'), (zap_a, 'zap', 'zap') )
```

The Analysis objects `foo_a`, `bar_a`, and `zap_a` are modified so that the latter two refer to the first for common dependencies.

Following this you can copy the `PYZ`, `EXE` and `COLLECT` statements from the original three spec files, substituting the unique names of the Analysis objects where the original spec files have `a.`, for example:

```
foo_pyz = PYZ(foo_a.pure)
foo_exe = EXE(foo_pyz, foo_a.scripts, ... etc.
foo_coll = COLLECT( foo_exe, foo_a.binaries, foo_a.datas... etc.

bar_pyz = PYZ(bar_a.pure)
bar_exe = EXE(bar_pyz, bar_a.scripts, ... etc.
bar_coll = COLLECT( bar_exe, bar_a.binaries, bar_a.datas... etc.
```

(If you are building one-file apps, there is no `COLLECT` step.) Save the combined spec file as `foobarzap.spec` and then build it:

```
pyi-build foobarzap.spec
```

The output in the `dist` folder will be all three apps, but the apps `dist/bar/bar` and `dist/zap/zap` will refer to the contents of `dist/foo/` for shared dependencies.

There are several multipackage examples in the *PyInstaller* distribution folder under `/tests/old_suite/multipackage`.

Remember that a spec file is executable Python. You can use all the Python facilities (`for` and `with` and the members of `sys` and `io`) in creating the Analysis objects and performing the `PYZ`, `EXE` and `COLLECT` statements. You may also need to know and use *The TOC and Tree Classes* described below.

Globals Available to the Spec File

While a spec file is executing it has access to a limited set of global names. These names include the classes defined by *PyInstaller*: `Analysis`, `BUNDLE`, `COLLECT`, `EXE`, `MERGE`, `PYZ`, `TOC` and `Tree`, which are discussed in the preceding sections.

Other globals contain information about the build environment:

DISTPATH The relative path to the `dist` folder where the application will be stored. The default path is relative to the current directory. If the `--distpath=` option is used, `DISTPATH` contains that value.

HOMEPATH The absolute path to the *PyInstaller* distribution, typically in the current Python site-packages folder.

SPEC The complete spec file argument given to the `pyinstaller` command, for example `myscript.spec` or `source/myscript.spec`.

SPECPATH The path prefix to the `SPEC` value as returned by `os.path.split()`.

specnm The name of the spec file, for example `myscript`.

workpath The path to the build directory. The default is relative to the current directory. If the `workpath=` option is used, `workpath` contains that value.

WARNFILE The full path to the warnings file in the build directory, for example `build/warnmyscript.txt`.

When Things Go Wrong

The information above covers most normal uses of *PyInstaller*. However, the variations of Python and third-party libraries are endless and unpredictable. It may happen that when you attempt to bundle your app either *PyInstaller* it-

self, or your bundled app, terminates with a Python traceback. Then please consider the following actions in sequence, before asking for technical help.

Recipes and Examples for Specific Problems

The *PyInstaller FAQ* page has work-arounds for some common problems. Code examples for some advanced uses and some common problems are available on our [PyInstaller Recipes](#) page. Some of the recipes there include:

- A more sophisticated way of collecting data files than the one shown above (*Adding Files to the Bundle*).
- Bundling a typical Django app.
- A use of a run-time hook to set the PyQt4 API level.
- A workaround for a multiprocessing constraint under Windows.

and others. Many of these Recipes were contributed by users. Please feel free to contribute more recipes!

Finding out What Went Wrong

Build-time Messages

When the `Analysis` step runs, it produces error and warning messages. These display after the command line if the `--log-level` option allows it. `Analysis` also puts messages in a warnings file named `build/name/warnname.txt` in the `work-path=` directory.

`Analysis` creates a message when it detects an import and the module it names cannot be found. A message may also be produced when a class or function is declared in a package (an `__init__.py` module), and the import specifies `package.name`. In this case, the analysis can't tell if `name` is supposed to refer to a submodule or package.

The “module not found” messages are not classed as errors because typically there are many of them. For example, many standard modules conditionally import modules for different platforms that may or may not be present.

All “module not found” messages are written to the `build/name/warnname.txt` file. They are not displayed to standard output because there are many of them. Examine the warning file; often there will be dozens of modules not found, but their absence has no effect.

When you run the bundled app and it terminates with an `ImportError`, that is the time to examine the warning file. Then see *Helping PyInstaller Find Modules* below for how to proceed.

Build-Time Dependency Graph

If you specify `--log-level=DEBUG` to the `pyinstaller` command, *PyInstaller* writes two files of data about dependencies into the build folder.

The file `build/name/xref-name.html` in the `work-path=` directory is an HTML file that lists the full contents of the import graph, showing which modules are imported by which. You can open it in any web browser. Find a module name, then keep clicking the “imported by” links until you find the top-level import that causes that module to be included.

The file `build/name/graph-name.dot` in the `work-path=` directory is a [GraphViz](#) input file. You can process it with the [GraphViz](#) command `dot` to produce a graphical display of the import dependencies.

These files are very large because even the simplest “hello world” Python program ends up including a large number of standard modules. For this reason the graph file is not very useful in this release.

Build-Time Python Errors

PyInstaller sometimes terminates by raising a Python exception. In most cases the reason is clear from the exception message, for example “Your system is not supported”, or “Pyinstaller requires at least Python 2.7”. Others clearly indicate a bug that should be reported.

One of these errors can be puzzling, however: `IOError("Python library not found!")` *PyInstaller* needs to bundle the Python library, which is the main part of the Python interpreter, linked as a dynamic load library. The name and location of this file varies depending on the platform in use. Some Python installations do not include a dynamic Python library by default (a static-linked one may be present but cannot be used). You may need to install a development package of some kind. Or, the library may exist but is not in a folder where *PyInstaller* is searching.

The places where *PyInstaller* looks for the python library are different in different operating systems, but `/lib` and `/usr/lib` are checked in most systems. If you cannot put the python library there, try setting the correct path in the environment variable `LD_LIBRARY_PATH` in Linux or `DYLD_LIBRARY_PATH` in OS X.

Getting Debug Messages

Giving the `--debug` option causes the bundled executable itself to write progress messages when it runs. This can be useful during development of a complex package, or when your app doesn't seem to be starting, or just to learn how the runtime works.

Normally the debug progress messages go to standard output. If the `--windowed` option is used when bundling a Windows app, they are displayed as `MessageBoxes`. For a `--windowed` Mac OS app they are not displayed.

Remember to bundle without `--debug` for your production version. Users would find the messages annoying.

Getting Python's Verbose Imports

You can also pass a `-v` (verbose imports) flag to the embedded Python interpreter (see [Giving Run-time Python Options](#) above). This can be extremely useful. It can be informative even with apps that are apparently working, to make sure that they are getting all imports from the bundle, and not leaking out to the local installed Python.

Python verbose and warning messages always go to standard output and are not visible when the `--windowed` option is used. Remember to not use this in the distributed program.

Helping PyInstaller Find Modules

Extending the Path

If Analysis recognizes that a module is needed, but cannot find that module, it is often because the script is manipulating `sys.path`. The easiest thing to do in this case is to use the `--paths=` option to list all the other places that the script might be searching for imports:

```
pyi-makespec --paths=/path/to/thisdir \  
             --paths=/path/to/otherdir myscript.py
```

These paths will be noted in the spec file. They will be added to the current `sys.path` during analysis.

Listing Hidden Imports

If Analysis thinks it has found all the imports, but the app fails with an import error, the problem is a hidden import; that is, an import that is not visible to the analysis phase.

Hidden imports can occur when the code is using `__import__` or perhaps `exec` or `eval`. Hidden imports can also occur when an extension module uses the Python/C API to do an import. When this occurs, Analysis can detect nothing. There will be no warnings, only an `ImportError` at run-time.

To find these hidden imports, build the app with the `-v` flag (*Getting Python's Verbose Imports* above) and run it.

Once you know what modules are needed, you add the needed modules to the bundle using the `--hidden-import=` command option, or by editing the spec file, or with a hook file (see *Understanding PyInstaller Hooks* below).

Extending a Package's `__path__`

Python allows a script to extend the search path used for imports through the `__path__` mechanism. Normally, the `__path__` of an imported module has only one entry, the directory in which the `__init__.py` was found. But `__init__.py` is free to extend its `__path__` to include other directories. For example, the `win32com.shell.shell` module actually resolves to `win32com/win32comext/shell/shell.pyd`. This is because `win32com/__init__.py` appends `../win32comext` to its `__path__`.

Because the `__init__.py` of an imported module is not actually executed during analysis, changes it makes to `__path__` are not seen by *PyInstaller*. We fix the problem with the same hook mechanism we use for hidden imports, with some additional logic; see *Understanding PyInstaller Hooks* below.

Note that manipulations of `__path__` hooked in this way apply only to the Analysis. At runtime all imports are intercepted and satisfied from within the bundle. `win32com.shell` is resolved the same way as `win32com.anythingelse`, and `win32com.__path__` knows nothing of `../win32comext`.

Once in a while, that's not enough.

Changing Runtime Behavior

More bizarre situations can be accommodated with runtime hooks. These are small scripts that manipulate the environment before your main script runs, effectively providing additional top-level code to your script.

There are two ways of providing runtime hooks. You can name them with the option `--runtime-hook=path-to-script`.

Second, some runtime hooks are provided. At the end of an analysis, the names in the module list produced by the Analysis phase are looked up in `loader/rthooks.dat` in the *PyInstaller* install folder. This text file is the string representation of a Python dictionary. The key is the module name, and the value is a list of hook-script pathnames. If there is a match, those scripts are included in the bundled app and will be called before your main script starts.

Hooks you name with the option are executed in the order given, and before any installed runtime hooks. If you specify `--runtime-hook=file1.py --runtime-hook=file2.py` then the execution order at runtime will be:

1. Code of `file1.py`.
2. Code of `file2.py`.
3. Any hook specified for an included module that is found in `rthooks/rthooks.dat`.
4. Your main script.

Hooks called in this way, while they need to be careful of what they import, are free to do almost anything. One reason to write a run-time hook is to override some functions or variables from some modules. A good example of this is the Django runtime hook (see `loader/rthooks/pyi_rth_django.py` in the *PyInstaller* folder). Django imports some modules dynamically and it is looking for some `.py` files. However `.py` files are not available in the one-file bundle. We need to override the function `django.core.management.find_commands` in a way that will just return a list of values. The runtime hook does this as follows:

```
import django.core.management
def _find_commands(_):
    return """cleanup shell runfcgi runserver""".split()
django.core.management.find_commands = _find_commands
```

Getting the Latest Version

If you have some reason to think you have found a bug in *PyInstaller* you can try downloading the latest development version. This version might have fixes or features that are not yet at PyPI. You can download the latest stable version and the latest development version from the [PyInstaller Downloads](#) page.

You can also install the latest version of *PyInstaller* directly using `pip`:

```
pip install -e https://github.com/pyinstaller/pyinstaller/archive/develop.zip
```

Asking for Help

When none of the above suggestions help, do ask for assistance on the [PyInstaller Email List](#).

Then, if you think it likely that you see a bug in *PyInstaller*, refer to the [How to Report Bugs](#) page.

Advanced Topics

The following discussions cover details of *PyInstaller* internal methods. You should not need this level of detail for normal use, but such details are helpful if you want to investigate the *PyInstaller* code and possibly contribute to it, as described in [How to Contribute](#).

The Bootstrap Process in Detail

There are many steps that must take place before the bundled script can begin execution. A summary of these steps was given in the Overview ([How the One-Folder Program Works](#) and [How the One-File Program Works](#)). Here is more detail to help you understand what the bootloader does and how to figure out problems.

Bootloader

The bootloader prepares everything for running Python code. It begins the setup and then returns itself in another process. This approach of using two processes allows a lot of flexibility and is used in all bundles except one-folder mode in Windows. So do not be surprised if you will see your bundled app as two processes in your system task manager.

What happens during execution of bootloader:

1. First process: bootloader starts.
 - (a) If one-file mode, extract bundled files to `temppath_MEIxxxxxx`
 - (b) Modify various environment variables:
 - Linux: save original value of `LD_LIBRARY_PATH` into `LD_LIBRARY_PATH_ORIG`, prepend our path to `LD_LIBRARY_PATH`.
 - AIX: same thing, but using `LIBPATH` and `LIBPATH_ORIG`.

- OSX: unset DYLD_LIBRARY_PATH.
- (c) Set up to handle signals for both processes.
 - (d) Run the child process.
 - (e) Wait for the child process to finish.
 - (f) If one-file mode, delete `temppath_MEIxxxxx`.
2. Second process: bootloader itself started as a child process.
 - (a) On Windows set the `activation context`.
 - (b) Load the Python dynamic library. The name of the dynamic library is embedded in the executable file.
 - (c) Initialize Python interpreter: set `sys.path`, `sys.prefix`, `sys.executable`.
 - (d) Run python code.

Running Python code requires several steps:

1. Run the Python initialization code which prepares everything for running the user's main script. The initialization code can use only the Python built-in modules because the general import mechanism is not yet available. It sets up the Python import mechanism to load modules only from archives embedded in the executable. It also adds the attributes `frozen` and `_MEIPASS` to the `sys` built-in module.
2. Execute any run-time hooks: first those specified by the user, then any standard ones.
3. Install python "egg" files. When a module is part of a zip file (.egg), it has been bundled into the `./eggs` directory. Installing means appending .egg file names to `sys.path`. Python automatically detects whether an item in `sys.path` is a zip file or a directory.
4. Run the main script.

Python imports in a bundled app

PyInstaller embeds compiled python code (.pyc files) within the executable. *PyInstaller* injects its code into the normal Python import mechanism. Python allows this; the support is described in [PEP 302](#) "New Import Hooks".

PyInstaller implements the PEP 302 specification for importing built-in modules, importing "frozen" modules (compiled python code bundled with the app) and for C-extensions. The code can be read in `./PyInstaller/loader/pyi_mod03_importers.py`.

At runtime the *PyInstaller* [PEP 302](#) hooks are appended to the variable `sys.meta_path`. When trying to import modules the interpreter will first try PEP 302 hooks in `sys.meta_path` before searching in `sys.path`. As a result, the Python interpreter loads imported python modules from the archive embedded in the bundled executable.

This is the resolution order of import statements in a bundled app:

1. Is it a built-in module? A list of built-in modules is in variable `sys.builtin_module_names`.
2. Is it a module embedded in the executable? Then load it from embedded archive.
3. Is it a C-extension? The app will try to find a file with name `package.subpackage.module.pyd` or `package.subpackage.module.so`
4. Next examine paths in the `sys.path`. There could be any additional location with python modules or .egg filenames.
5. If the module was not found then raise `ImportError`.

The TOC and Tree Classes

PyInstaller manages lists of files using the TOC (Table Of Contents) class. It provides the `Tree` class as a convenient way to build a TOC from a folder path.

TOC Class (Table of Contents)

Objects of the TOC class are used as input to the classes created in a spec file. For example, the `scripts` member of an Analysis object is a TOC containing a list of scripts. The `pure` member is a TOC with a list of modules, and so on.

Basically a TOC object contains a list of tuples of the form

```
(name, path, typecode)
```

In fact, it acts as an ordered set of tuples; that is, it contains no duplicates (where uniqueness is based on the *name* element of each tuple). Within this constraint, a TOC preserves the order of tuples added to it.

A TOC behaves like a list and supports the same methods such as appending, indexing, etc. A TOC also behaves like a set, and supports taking differences and intersections. In all of these operations a list of tuples can be used as one argument. For example, the following expressions are equivalent ways to add a file to the `a.datas` member:

```
a.datas.append( [ ('README', 'src/README.txt', 'DATA' ) ] )
a.datas += [ ('README', 'src/README.txt', 'DATA' ) ]
```

Set-difference makes excluding modules quite easy. For example:

```
a.binaries - [('badmodule', None, None)]
```

is an expression that produces a new TOC that is a copy of `a.binaries` from which any tuple named `badmodule` has been removed. The right-hand argument to the subtraction operator is a list that contains one tuple in which *name* is `badmodule` and the *path* and *typecode* elements are `None`. Because set membership is based on the *name* element of a tuple only, it is not necessary to give accurate *path* and *typecode* elements when subtracting.

In order to add files to a TOC, you need to know the *typecode* values and their related *path* values. A *typecode* is a one-word string. *PyInstaller* uses a number of *typecode* values internally, but for the normal case you need to know only these:

| typecode | description | name | path |
|-------------|-------------------------------|----------------|--------------------------|
| 'DATA' | Arbitrary files. | Run-time name. | Full path name in build. |
| 'BINARY' | A shared library. | Run-time name. | Full path name in build. |
| 'EXTENSION' | A binary extension to Python. | Run-time name. | Full path name in build. |
| 'OPTION' | A Python run-time option. | Option code | ignored. |

The run-time name of a file will be used in the final bundle. It may include path elements, for example `extras/mydata.txt`.

A `BINARY` file or an `EXTENSION` file is assumed to be loadable, executable code, for example a dynamic library. The types are treated the same. `EXTENSION` is generally used for a Python extension module, for example a module compiled by `Cython`. *PyInstaller* will examine either type of file for dependencies, and if any are found, they are also included.

The Tree Class

The `Tree` class is a way of creating a TOC that describes some or all of the files within a directory:

```
Tree(root, prefix=run-time-folder, excludes=string_list, typecode=code|'DATA' )
```

- The *root* argument is a path string to a directory. It may be absolute or relative to the spec file directory.

- The *prefix* argument, if given, is a name for a subfolder within the run-time folder to contain the tree files. If you omit *prefix* or give `None`, the tree files will be at the top level of the run-time folder.
- The *excludes* argument, if given, is a list of one or more strings that match files in the *root* that should be omitted from the Tree. An item in the list can be either:
 - a name, which causes files or folders with this basename to be excluded
 - `*.ext`, which causes files with this extension to be excluded
- The *typecode* argument, if given, specifies the TOC typecode string that applies to all items in the Tree. If omitted, the default is `DATA`, which is appropriate for most cases.

For example:

```
extras_toc = Tree('../src/extras', prefix='extras', excludes=['tmp', '*.pyc'])
```

This creates `extras_toc` as a TOC object that lists all files from the relative path `../src/extras`, omitting those that have the basename (or are in a folder named) `tmp` or that have the type `.pyc`. Each tuple in this TOC has:

- A *name* composed of `extras/filename`.
- A *path* consisting of a complete, absolute path to that file in the `../src/extras` folder (relative to the location of the spec file).
- A *typecode* of `DATA` (by default).

An example of creating a TOC listing some binary modules:

```
cython_mods = Tree( '../src/cy_mods', excludes=['*.pyx', '*.py', '*.pyc'], typecode=
↳ 'EXTENSION' )
```

This creates a TOC with a tuple for every file in the `cy_mods` folder, excluding any with the `.pyx`, `.py` or `.pyc` suffixes (so presumably collecting the `.pyd` or `.so` modules created by Cython). Each tuple in this TOC has:

- Its own filename as *name* (no prefix; the file will be at the top level of the bundle).
- A *path* as an absolute path to that file in `../src/cy_mods` relative to the spec file.
- A *typecode* of `EXTENSION` (`BINARY` could be used as well).

Inspecting Archives

An archive is a file that contains other files, for example a `.tar` file, a `.jar` file, or a `.zip` file. Two kinds of archives are used in *PyInstaller*. One is a `ZlibArchive`, which allows Python modules to be stored efficiently and, with some import hooks, imported directly. The other, a `CArchive`, is similar to a `.zip` file, a general way of packing up (and optionally compressing) arbitrary blobs of data. It gets its name from the fact that it can be manipulated easily from C as well as from Python. Both of these derive from a common base class, making it fairly easy to create new kinds of archives.

ZlibArchive

A `ZlibArchive` contains compressed `.pyc` or `.pyo` files. The `PYZ` class invocation in a spec file creates a `ZlibArchive`.

The table of contents in a `ZlibArchive` is a Python dictionary that associates a key, which is a member's name as given in an `import` statement, with a seek position and a length in the `ZlibArchive`. All parts of a `ZlibArchive` are stored in the `marshalled` format and so are platform-independent.

A ZlibArchive is used at run-time to import bundled python modules. Even with maximum compression this works faster than the normal import. Instead of searching `sys.path`, there's a lookup in the dictionary. There are no directory operations and no file to open (the file is already open). There's just a seek, a read and a decompress.

A Python error trace will point to the source file from which the archive entry was created (the `__file__` attribute from the time the `.pyc` was compiled, captured and saved in the archive). This will not tell your user anything useful, but if they send you a Python error trace, you can make sense of it.

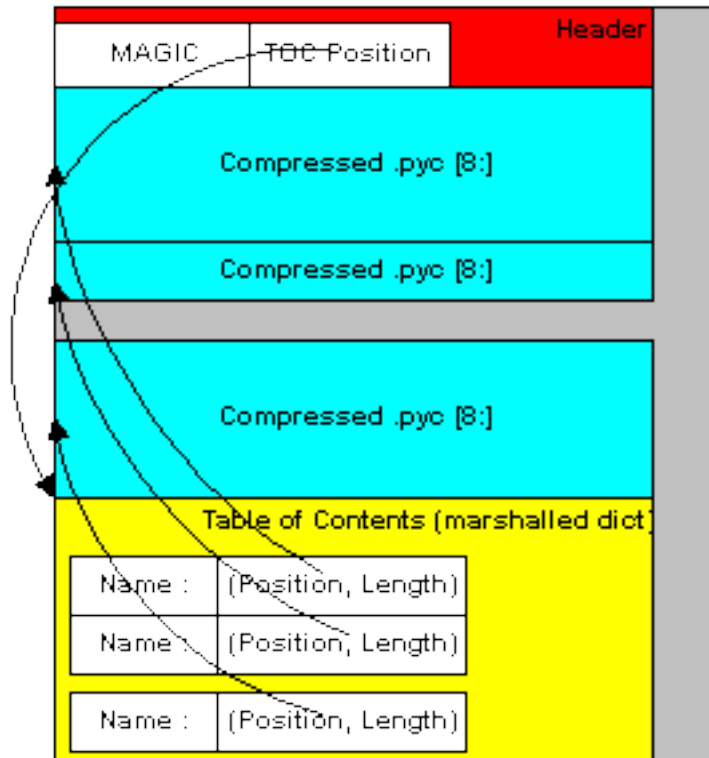


Fig. 1.1: Structure of the ZlibArchive

CArchive

A CArchive can contain any kind of file. It's very much like a `.zip` file. They are easy to create in Python and easy to unpack from C code. A CArchive can be appended to another file, such as an ELF and COFF executable. To allow this, the archive is made with its table of contents at the end of the file, followed only by a cookie that tells where the table of contents starts and where the archive itself starts.

A CArchive can be embedded within another CArchive. An inner archive can be opened and used in place, without having to extract it.

Each table of contents entry has variable length. The first field in the entry gives the length of the entry. The last field is the name of the corresponding packed file. The name is null terminated. Compression is optional for each member.

There is also a type code associated with each member. The type codes are used by the self-extracting executables. If you're using a CArchive as a `.zip` file, you don't need to worry about the code.

The ELF executable format (Windows, Linux and some others) allows arbitrary data to be concatenated to the end of the executable without disturbing its functionality. For this reason, a CArchive's Table of Contents is at the end of the archive. The executable can open itself as a binary file, seek to the end and 'open' the CArchive.

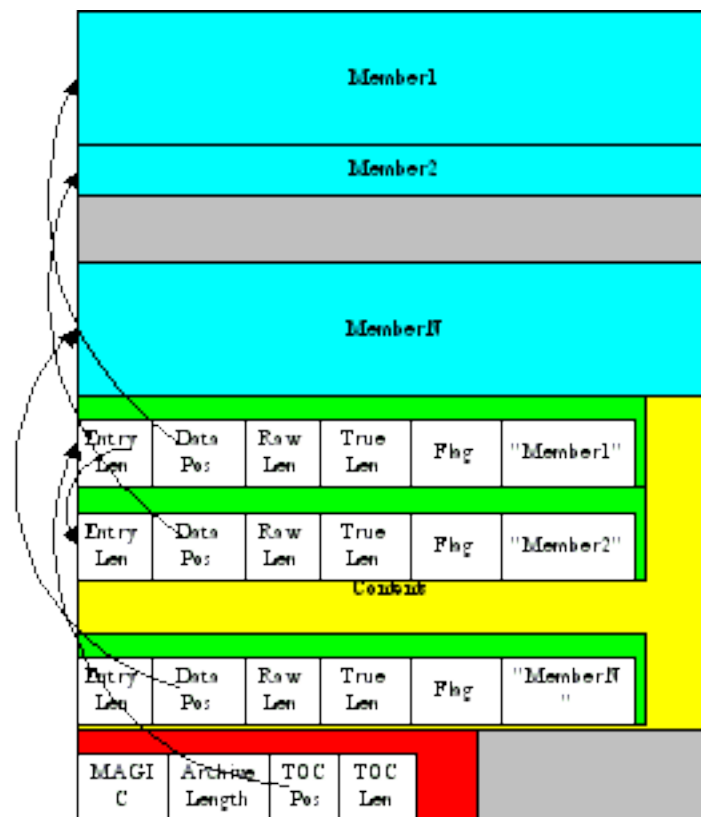


Fig. 1.2: Structure of the CArchive

Using pyi-archive_viewer

Use the `pyi-archive_viewer` command to inspect any type of archive:

```
pyi-archive_viewer archivefile
```

With this command you can examine the contents of any archive built with *PyInstaller* (a PYZ or PKG), or any executable (.exe file or an ELF or COFF binary). The archive can be navigated using these commands:

O name Open the embedded archive *name* (will prompt if omitted). For example when looking in a one-file executable, you can open the `outPYZ.pyz` archive inside it.

U Go up one level (back to viewing the containing archive).

X name Extract *name* (will prompt if omitted). Prompts for an output filename. If none given, the member is extracted to stdout.

Q Quit.

The `pyi-archive_viewer` command has these options:

```
-h, --help    Show help.
```

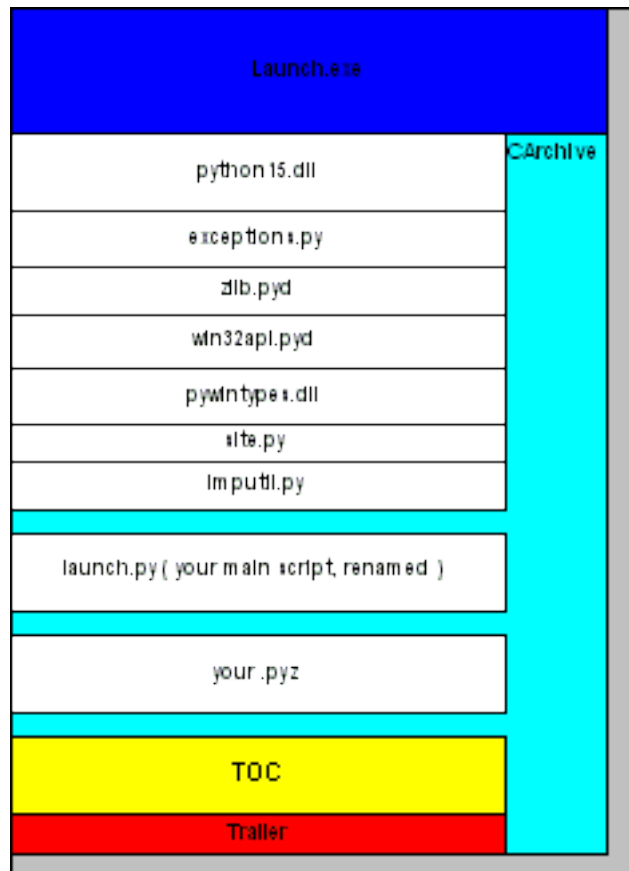


Fig. 1.3: Structure of the Self Extracting Executable

| | |
|------------------------|--|
| -l, --log | Quick contents log. |
| -b, --brief | Print a python evaluable list of contents filenames. |
| -r, --recursive | Used with -l or -b, applies recursive behaviour. |

Inspecting Executables

You can inspect any executable file with `pyi-bindepend`:

```
pyi-bindepend executable_or_dynamic_library
```

The `pyi-bindepend` command analyzes the executable or DLL you name and writes to stdout all its binary dependencies. This is handy to find out which DLLs are required by an executable or by another DLL.

`pyi-bindepend` is used by *PyInstaller* to follow the chain of dependencies of binary extensions during Analysis.

Creating a Reproducible Build

In certain cases it is important that when you build the same application twice, using exactly the same set of dependencies, the two bundles should be exactly, bit-for-bit identical.

That is not the case normally. Python uses a random hash to make dicts and other hashed types, and this affects compiled byte-code as well as *PyInstaller* internal data structures. As a result, two builds may not produce bit-for-bit identical results even when all the components of the application bundle are the same and the two applications execute in identical ways.

You can assure that a build will produce the same bits by setting the `PYTHONHASHSEED` environment variable to a known integer value before running *PyInstaller*. This forces Python to use the same random hash sequence until `PYTHONHASHSEED` is unset or set to `'random'`. For example, execute *PyInstaller* in a script such as the following (for Linux and OS X):

```
# set seed to a known repeatable integer value
PYTHONHASHSEED=1
export PYTHONHASHSEED
# create one-file build as myscript
pyinstaller myscript.spec
# make checksum
cksum dist/myscript/myscript | awk '{print $1}' > dist/myscript/checksum.txt
# let Python be unpredictable again
unset PYTHONHASHSEED
```

Understanding PyInstaller Hooks

In summary, a “hook” file extends *PyInstaller* to adapt it to the special needs and methods used by a Python package. The word “hook” is used for two kinds of files. A *runtime* hook helps the bootloader to launch an app. For more on runtime hooks, see [Changing Runtime Behavior](#). Other hooks run while an app is being analyzed. They help the Analysis phase find needed files.

The majority of Python packages use normal methods of importing their dependencies, and *PyInstaller* locates all their files without difficulty. But some packages make unusual uses of the Python import mechanism, or make clever changes to the import system at runtime. For this or other reasons, *PyInstaller* cannot reliably find all the needed files, or may include too many files. A hook can tell about additional source files or data files to import, or files not to import.

A hook file is a Python script, and can use all Python features. It can also import helper methods from `PyInstaller.utils.hooks` and useful variables from `PyInstaller.compat`. These helpers are documented below.

The name of a hook file is `hook-full-import-name.py`, where *full-import-name* is the fully-qualified name of an imported script or module. You can browse through the existing hooks in the `hooks` folder of the *PyInstaller* distribution folder and see the names of the packages for which hooks have been written. For example `hook-PyQt5.QtCore.py` is a hook file telling about hidden imports needed by the module `PyQt5.QtCore`. When your script contains `import PyQt5.QtCore` (or `from PyQt5 import QtCore`), Analysis notes that `hook-PyQt5.QtCore.py` exists, and will call it.

Many hooks consist of only one statement, an assignment to `hiddenimports`. For example, the hook for the `dnspython` package, called `hook-dns.rdata.py`, has only this statement:

```
hiddenimports = [
    "dns.rdtypes.*",
    "dns.rdtypes.ANY.*"
]
```

When Analysis sees `import dns.rdata` or `from dns import rdata` it calls `hook-dns.rdata.py` and examines its value of `hiddenimports`. As a result, it is as if your source script also contained:

```
import dns.rdtypes.*
import dns.rdtypes.ANY.*
```

A hook can also cause the addition of data files, and it can cause certain files to *not* be imported. Examples of these actions are shown below.

When the module that needs these hidden imports is useful only to your project, store the hook file(s) somewhere near your source file. Then specify their location to the `pyinstaller` or `pyi-makespec` command with the `--additional-hooks-dir=` option. If the hook file(s) are at the same level as the script, the command could be simply:

```
pyinstaller --additional-hooks-dir=. myscript.py
```

If you write a hook for a module used by others, please send us the hook file so we can make it available.

How a Hook Is Loaded

A hook is a module named `hook-full-import-name.py` in a folder where the Analysis object looks for hooks. Each time Analysis detects an import, it looks for a hook file with a matching name. When one is found, Analysis imports the hook's code into a Python namespace. This results in the execution of all top-level statements in the hook source, for example import statements, assignments to global names, and function definitions. The names defined by these statements are visible to Analysis as attributes of the namespace.

Thus a hook is a normal Python script and can use all normal Python facilities. For example it could test `sys.version` and adjust its assignment to `hiddenimports` based on that. There are over 150 hooks in the *PyInstaller* installation. You are welcome to browse through them for examples.

Hook Global Variables

A majority of the existing hooks consist entirely of assignments of values to one or more of the following global variables. If any of these are defined by the hook, Analysis takes their values and applies them to the bundle being created.

hiddenimports A list of module names (relative or absolute) that should be part of the bundled app. This has the same effect as the `--hidden-import` command line option, but it can contain a list of names and is applied automatically only when the hooked module is imported. Example:

```
hiddenimports = ['_proxy', 'utils', 'defs']
```

excludedimports A list of absolute module names that should *not* be part of the bundled app. If an excluded module is imported only by the hooked module or one of its sub-modules, the excluded name and its sub-modules will not be part of the bundle. (If an excluded name is explicitly imported in the source file or some other module, it will be kept.) Several hooks use this to prevent automatic inclusion of the `tkinter` module. Example:

```
excludedimports = [modname_tkinter]
```

datas A list of files to bundle with the app as data. Each entry in the list is a tuple containing two strings. The first string specifies a file (or file “glob”) in this system, and the second specifies the name(s) the file(s) are to have in the bundle. (This is the same format as used for the `datas=` argument, see [Adding Data Files](#).) Example:

```
datas = [ ('/usr/share/icons/education_*.png', 'icons') ]
```

If you need to collect multiple directories or nested directories, you can use helper functions from the `PyInstaller.utils.hooks` module (see below) to create this list, for example:

```
datas = collect_data_files('submodule1')
datas+= collect_data_files('submodule2')
```

In rare cases you may need to apply logic to locate particular files within the file system, for example because the files are in different places on different platforms or under different versions. Then you can write a `hook()` function as described below under [The hook\(hook_api\) Function](#).

binaries A list of files or directories to bundle as binaries. The format is the same as `datas` (tuples with strings that specify the source and the destination). Binaries is a special case of `datas`, in that PyInstaller will check each file to see if it depends on other dynamic libraries. Example:

```
binaries = [ ('C:\\Windows\\System32\\*.dll', 'dlls') ]
```

Many hooks use helpers from the `PyInstaller.utils.hooks` module to create this list (see below):

```
binaries = collect_dynamic_libs('zmq')
```

Useful Items in `PyInstaller.compat`

A hook may import the following names from `PyInstaller.compat`, for example:

```
from PyInstaller.compat import modname_tkinter, is_win
```

is_py2: True when the active Python is version 2.7.

is_py3: True when the active Python is version 3.X.

is_py34, is_py35, is_py36: True when the current version of Python is at least 3.4, 3.5 or 3.6 respectively.

is_win: True in a Windows system.

is_cygwin: True when `sys.platform=='cygwin'`.

is_darwin: True in Mac OS X.

is_linux: True in any Linux system (`sys.platform.startswith('linux')`).

is_solaris: True in Solaris.

is_aix: True in AIX.

is_freebsd: True in FreeBSD.

is_venv: True in any virtual environment (either `virtualenv` or `venv`).

base_prefix: String, the correct path to the base Python installation, whether the installation is native or a virtual environment.

modname_tkinter: String, `Tkinter` in Python 2.7 but `tkinter` in Python 3. To prevent an unnecessary import of `Tkinter`, write:

```
from PyInstaller.compat import modname_tkinter
excludedimports = [ modname_tkinter ]
```

EXTENSION_SUFFIXES: List of Python C-extension file suffixes. Used for finding all binary dependencies in a folder; see `hook-cryptography.py` for an example.

Useful Items in `PyInstaller.utils.hooks`

A hook may import useful functions from `PyInstaller.utils.hooks`. Use a fully-qualified import statement, for example:

```
from PyInstaller.utils.hooks import collect_data_files, eval_statement
```

The `PyInstaller.utils.hooks` functions listed here are generally useful and used in a number of existing hooks. There are several more functions besides these that serve the needs of specific hooks, such as hooks for `PyQt4/5`. You are welcome to read the `PyInstaller.utils.hooks` module (and read the existing hooks that import from it) to get code and ideas.

exec_statement('statement'): Execute a single Python statement in an externally-spawned interpreter and return the standard output that results, as a string. Examples:

```
tk_version = exec_statement(
    "from _tkinter import TK_VERSION; print(TK_VERSION)"
)

mpl_data_dir = exec_statement(
    "import matplotlib; print(matplotlib._get_data_path())"
)

datas = [ (mpl_data_dir, "") ]
```

eval_statement('statement'): Execute a single Python statement in an externally-spawned interpreter. If the resulting standard output text is not empty, apply the `eval()` function to it; else return `None`. Example:

```
databases = eval_statement('''
import sqlalchemy.databases
print(sqlalchemy.databases.__all__)
''')

for db in databases:
    hiddenimports.append("sqlalchemy.databases." + db)
```

is_module_satisfies(requirements, version=None, version_attr='__version__'): Check that the named module (fully-qualified) exists and satisfies the given requirement. Example:

```
if is_module_satisfies('sqlalchemy >= 0.6'):
```

This function provides robust version checking based on the same low-level algorithm used by `easy_install` and `pip`, and should always be used in preference to writing your own comparison code. In particular, version strings should never be compared lexicographically (except for exact equality). For example `'00.5' > '0.6'` returns `True`, which is not the desired result.

The `requirements` argument uses the same syntax as supported by the `Package resources` module of `setup tools` (follow the link to see the supported syntax).

The optional `version` argument is a PEP0440-compliant, dot-delimited version specifier such as `'3.14-rc5'`.

When the package being queried has been installed by `easy_install` or `pip`, the existing `setup tools` machinery is used to perform the test and the `version` and `version_attr` arguments are ignored.

When that is not the case, the `version` argument is taken as the installed version of the package (perhaps obtained by interrogating the package in some other way). When `version` is `None`, the named package is imported into a subprocess, and the `__version__` value of that import is tested. If the package uses some other name than `__version__` for its version global, that name can be passed as the `version_attr` argument.

For more details and examples refer to the function's doc-string, found in `Pyinstaller/utils/hooks/__init__.py`.

`collect_submodules('package-name', pattern=None)`: Returns a list of strings that specify all the modules in a package, ready to be assigned to the `hiddenimports` global. Returns an empty list when package does not name a package (a package is defined as a module that contains a `__path__` attribute).

The `pattern`, if given, is function to filter through the submodules found, selecting which should be included in the returned list. It takes one argument, a string, which gives the name of a submodule. Only if the function returns `true` is the given submodule is added to the list of returned modules. For example, `filter=lambda name: 'test' not in name` will return modules that don't contain the word `test`.

`is_module_or_submodule(name, mod_or_submod)`: This helper function is designed for use in the `filter` argument of `collect_submodules`, by returning `True` if the given name is a module or a submodule of `mod_or_submod`. For example: `collect_submodules('foo', lambda name: not is_module_or_submodule(name, 'foo.test'))` excludes `foo.test` and `foo.test.one` but not `foo.testifier`.

`collect_data_files('module-name', subdir=None, include_py_files=False)`: Returns a list of (source, dest) tuples for all non-Python (i.e. data) files found in `module-name`, ready to be assigned to the `datas` global. `module-name` is the fully-qualified name of a module or package (but not a zipped "egg"). The function uses `os.walk()` to visit the module directory recursively. `subdir`, if given, restricts the search to a relative subdirectory.

Normally Python executable files (ending in `.py`, `.pyc`, etc.) are not collected. Pass `include_py_files=True` to collect those files as well. (This can be used with routines such as those in `pkgutil` that search a directory for Python executable files and load them as extensions or plugins.)

`collect_dynamic_libs('module-name')`: Returns a list of (source, dest) tuples for all the dynamic libs present in a module directory. The list is ready to be assigned to the `binaries` global variable. The function uses `os.walk()` to examine all files in the module directory recursively. The name of each file found is tested against the likely patterns for a dynamic lib: `*.dll`, `*.dylib`, `lib*.pyd`, and `lib*.so`. Example:

```
binaries = collect_dynamic_libs( 'enchant' )
```

`get_module_file_attribute('module-name')`: Return the absolute path to `module-name`, a fully-qualified module name. Example:

```
nacl_dir = os.path.dirname(get_module_file_attribute('nacl'))
```

get_package_paths('package-name'): Given the name of a package, return a tuple. The first element is the absolute path to the folder where the package is stored. The second element is the absolute path to the named package. For example, if `pkg.subpkg` is stored in `/abs/Python/lib` the result of:

```
get_package_paths( 'pkg.subpkg' )
```

is the tuple, (`/abs/Python/lib`, `/abs/Python/lib/pkg/subpkg`)

copy_metadata('package-name'): Given the name of a package, return the name of its distribution metadata folder as a list of tuples ready to be assigned (or appended) to the `datas` global variable.

Some packages rely on metadata files accessed through the `pkg_resources` module. Normally *PyInstaller* does not include these metadata files. If a package fails without them, you can use this function in a hook file to easily add them to the bundle. The tuples in the returned list have two strings. The first is the full pathname to a folder in this system. The second is the folder name only. When these tuples are added to `datas`, the folder will be bundled at the top level. If *package-name* does not have metadata, an `AssertionError` exception is raised.

get_homebrew_path(formula=''): Return the homebrew path to the named formula, or to the global prefix when formula is omitted. Returns `None` if not found.

django_find_root_dir(): Return the path to the top-level Python package containing the Django files, or `None` if nothing can be found.

django_dottedstring_imports('django-root-dir') Return a list of all necessary Django modules specified in the Django settings.py file, such as the `Django.settings.INSTALLED_APPS` list and many others.

The hook (hook_api) Function

In addition to, or instead of, setting global values, a hook may define a function `hook(hook_api)`. A `hook()` function should only be needed if the hook needs to apply sophisticated logic or to make a complex search of the source machine.

The Analysis object calls the function and passes it a `hook_api` object which has the following immutable properties:

__name__: The fully-qualified name of the module that caused the hook to be called, e.g., `six.moves.tkinter`.

__file__: The absolute path of the module. If it is:

- A standard (rather than namespace) package, this is the absolute path of this package's directory.
- A namespace (rather than standard) package, this is the abstract placeholder `-`.
- A non-package module or C extension, this is the absolute path of the corresponding file.

__path__: A list of the absolute paths of all directories comprising the module if it is a package, or `None`. Typically the list contains only the absolute path of the package's directory.

The `hook_api` object also offers the following methods:

add_imports(*names): The `names` argument may be a single string or a list of strings giving the fully-qualified name(s) of modules to be imported. This has the same effect as adding the names to the `hiddenimports` global.

del_imports(*names): The `names` argument may be a single string or a list of strings, giving the fully-qualified name(s) of modules that are not to be included if they are imported only by the hooked module. This has the same effect as adding names to the `excludedimports` global.

add_datas(tuple_list): The `tuple_list` argument has the format used with the `datas` global variable. This call has the effect of adding items to that list.

add_binaries(tuple_list): The `tuple_list` argument has the format used with the `binaries` global variable. This call has the effect of adding items to that list.

The `hook()` function can add, remove or change included files using the above methods of `hook_api`. Or, it can simply set values in the four global variables, because these will be examined after `hook()` returns.

The `pre_find_module_path(pfmp_api)` Method

You may write a hook with the special function `pre_find_module_path(pfmp_api)`. This method is called when the hooked module name is first seen by Analysis, before it has located the path to that module or package (hence the name “pre-find-module-path”).

Hooks of this type are only recognized if they are stored in a sub-folder named `pre_find_module_path` in a hooks folder, either in the distributed hooks folder or an `--additional-hooks-dir` folder. You may have normal hooks as well as hooks of this type for the same module. For example *PyInstaller* includes both a `hooks/hook-distutils.py` and also a `hooks/pre_find_module_path/hook-distutils.py`.

The `pfmp_api` object that is passed has the following immutable attribute:

module_name: A string, the fully-qualified name of the hooked module.

The `pfmp_api` object has one mutable attribute, `search_dirs`. This is a list of strings that specify the absolute path, or paths, that will be searched for the hooked module. The paths in the list will be searched in sequence. The `pre_find_module_path()` function may replace or change the contents of `pfmp_api.search_dirs`.

Immediately after return from `pre_find_module_path()`, the contents of `search_dirs` will be used to find and analyze the module.

For an example of use, see the file `hooks/pre_find_module_path/hook-distutils.py`. It uses this method to redirect a search for `distutils` when *PyInstaller* is executing in a virtual environment.

The `pre_safe_import_module(psim_api)` Method

You may write a hook with the special function `pre_safe_import_module(psim_api)`. This method is called after the hooked module has been found, but *before* it and everything it recursively imports is added to the “graph” of imported modules. Use a pre-safe-import hook in the unusual case where:

- The script imports *package.dynamic-name*
- The *package* exists
- however, no module *dynamic-name* exists at compile time (it will be defined somehow at run time)

You use this type of hook to make dynamically-generated names known to PyInstaller. PyInstaller will not try to locate the dynamic names, fail, and report them as missing. However, if there are normal hooks for these names, they will be called.

Hooks of this type are only recognized if they are stored in a sub-folder named `pre_safe_import_module` in a hooks folder, either in the distributed hooks folder or an `--additional-hooks-dir` folder. (See the distributed `hooks/pre_safe_import_module` folder for examples.)

You may have normal hooks as well as hooks of this type for the same module. For example the distributed system has both a `hooks/hook-gi.repository.GLib.py` and also a `hooks/pre_safe_import_module/hook-gi.repository.GLib.py`.

The `psim_api` object offers the following attributes, all of which are immutable (an attempt to change one raises an exception):

module_basename: String, the unqualified name of the hooked module, for example `text`.

module_name: String, the fully-qualified name of the hooked module, for example `email.mime.text`.

module_graph: The module graph representing all imports processed so far.

parent_package: If this module is a top-level module of its package, `None`. Otherwise, the graph node that represents the import of the top-level module.

The last two items, `module_graph` and `parent_package`, are related to the module-graph, the internal data structure used by *PyInstaller* to document all imports. Normally you do not need to know about the module-graph.

The `psim_api` object also offers the following methods:

add_runtime_module(fully_qualified_name): Use this method to add an imported module whose name may not appear in the source because it is dynamically defined at run-time. This is useful to make the module known to *PyInstaller* and avoid misleading warnings. A typical use applies the name from the `psim_api`:

```
psim_api.add_runtime_module( psim_api.module_name )
```

add_alias_module(real_module_name, alias_module_name): `real_module_name` is the fully-qualified name of an existing module, one that has been or could be imported by name (it will be added to the graph if it has not already been imported). `alias_module_name` is a name that might be referenced in the source file but should be treated as if it were `real_module_name`. This method ensures that if *PyInstaller* processes an import of `alias_module_name` it will use `real_module_name`.

append_package_path(directory): The hook can use this method to add a package path to be searched by *PyInstaller*, typically an import path that the imported module would add dynamically to the path if the module was executed normally. `directory` is a string, a pathname to add to the `__path__` attribute.

Building the Bootloader

PyInstaller comes with pre-compiled bootloaders for some platforms in the `bootloader` folder of the distribution folder. When there is no pre-compiled bootloader, the `pip` setup will attempt to build one.

If there is no precompiled bootloader for your platform, or if you want to modify the bootloader source, you need to build the bootloader. To do this,

- `cd` into the distribution folder.
- `cd` `bootloader`.
- Make the bootloader with: `python ./waf distclean all`.

This will produce the bootloader executables,

- `./PyInstaller/bootloader/YOUR_OS/run`,
- `./PyInstaller/bootloader/YOUR_OS/run_d`
- `./PyInstaller/bootloader/YOUR_OS/runw` and
- `./PyInstaller/bootloader/YOUR_OS/runw_d`

Note: If you have multiple versions of Python, the Python you use to run `waf` is the one whose configuration is used.

If this reports an error, read the detailed notes that follow, then ask for technical help.

Development tools

On Debian/Ubuntu systems, you can run the following to install everything required:

```
sudo apt-get install build-essential
```

On Fedora/RHEL and derivatives, you can run the following:

```
su
yum groupinstall "Development Tools"
```

On Mac OS X you can get gcc by installing [Xcode](#). It is a suite of tools for developing software for Mac OS X. It can be also installed from your Mac OS X Install DVD. It is not necessary to install the version 4 of Xcode.

On Solaris and AIX the bootloader is built and tested with gcc.

Building for Windows

On Windows you can use the Visual Studio C++ compiler (Visual Studio 2008 is recommended). A free version you can download is [Visual Studio Express](#).

Note: When compiling libs to link with Python it is important to use the same level of Visual Studio as was used to compile Python. *That is not the case here.* The bootloader is a self-contained static executable that imposes no restrictions on the version of Python being used. So you can use any Visual Studio version that is convenient.

If Visual Studio is not convenient, you can download and install the MinGW distribution from one of the following locations:

- [MinGW-w64](#) required, uses gcc 4.4 and up.
- [TDM-GCC](#) - MinGW (not used) and MinGW-w64 installers

On Windows, when using MinGW-w64, add `PATH_TO_MINGW\bin` to your system `PATH` variable. Before building the bootloader run for example:

```
set PATH=C:\MinGW\bin;%PATH%
```

Change to the `bootloader` subdirectory. Run:

```
python ./waf distclean all
```

This will produce the bootloader executables `run*.exe` in the `.\PyInstaller\bootloader\YOUR_OS` directory.

Building Linux Standard Base (LSB) compliant binaries

By default, the bootloaders on Linux are "normal", non-LSB binaries, which should be fine for all GNU/Linux distributions.

If for some reason you want to build Linux Standard Base (LSB) compliant binaries⁰, you can do so by specifying `--lsb` on the `waf` command line, as follows:

⁰ Linux Standard Base (LSB) is a set of open standards that should increase compatibility among Linux distributions. Unfortunately it is not widely adopted and both Debian and Ubuntu dropped support for LSB in autumn 2015. Thus *PyInstaller* bootloaders are no longer provided as LSB binary.

```
python waf distclean all --lsb
```

LSB version 4.0 is required for successful building of bootloader. Please refer to `python waf --help` for further options related to LSB building.

The bootloaders will still end up in `support/loader/YOUR_OS/run`.

Changelog for PyInstaller

3.3 (unreleased)

- Add support for Python 3.6 (thanks to xoviat).
- (Bootloader) Linux bootloaders are now officially no LSB binaries. This was already the case since release 3.1, but documented the other way round. Also build defaults to non-LSB binaries now. (#2369)

3.2.1 (2017-01-15)

- New, updated and fixed hooks: botocore (#2094), gi (#2347), jira (#2222), PyQt5.QtWebEngineWidgets (#2269), skimage (#2195, 2225), sphinx (#2323,) xsge_gui (#2251).

Fixed the following issues:

- Don't fail if working directory already exists (#1994)
- Avoid encoding errors in main script (#1976)
- Fix hasher digest bytes not str (#2229, #2230)
- (Windows) Fix additional dependency on the msvcrt10.dll (#1974)
- (Windows) Correctly decode a bytes object produced by pefile (#1981)
- (Windows) Package `pefile` with `pyinstaller`. This partially undoes some changes in 3.2 in which the packaged pefiles were removed to use the pypi version instead. The pypi version was considerably slower in some applications, and still has a couple of small issues on PY3. (#1920)
- (OS X) PyQt5 packaging issues on MacOS (#1874)
- (OS X) Replace run-time search path keyword (#1965)
- (OS X) (Re-) add argv emulation for OSX, 64-bit (#2219)
- (OS X) use `decode("utf-8")` to convert bytes in `getImports_macholib()` (#1973)
- (Bootloader) fix segfaults (#2176)
- (setup.py) pass option `--no-lsb` on GNU/Linux only (#1975)
- Updates and fixes in documentation, manuals, et al. (#1986, 2002, #2153, #2227, #2231)

3.2 (2016-05-03)

- Even the “main” script is now byte-compiled (#1847, #1856)
- The manual is on `readthedocs.io` now (#1578)
- On installation try to compile the bootloader if there is none for the current platform (#1377)

- (Unix) Use `objcopy` to create a valid ELF file (#1812, #1831)
- (Linux): Compile with `_FORTIFY_SOURCE` (#1820)
- New, updated and fixed hooks: CherryPy (#1860), Cryptography (#1425, #1861), enchant (1562), `gi.repository.GdkPixbuf` (#1843), `gst` (#1963), `Lib2to3` (#1768), PyQt4, PyQt5, PySide (#1783, #1897, #1887), SciPy (#1908, #1909), sphinx (#1911, #1912), sqlalchemy (#1951), traitlets `wx.lib.pubsub` (#1837, #1838),
- For windowed mode add `isatty()` for our dummy `NullWriter` (#1883)
- Suppress “Failed to execute script” in case of `SystemExit` (#1869)
- Do not apply Upx compressor for bootloader files (#1863)
- Fix absolute path for lib used via `ctypes` (#1934)
- (OSX) Fix binary cache on NFS (#1573, #1849)
- (Windows) Fix message in `grab_version` (#1923)
- (Windows) Fix wrong icon paramter in Windows example (#1764)
- (Windows) Fix win32 unicode handling (#1878)
- (Windows) Fix unnecessary rebuilds caused by rebuilding `winmanifest` (#1933)
- (Cygwin) Fix finding the Python library for Cygwin 64-bit (#1307, #1810, #1811)
- (OSX) Fix compilation issue (#1882)
- (Windows) No longer bundle `pefile`, use package from pypi for windows (#1357)
- (Windows) Provide a more robust means of executing a Python script
- AIX fixes.
- Update `waf` to version 1.8.20 (#1868)
- Fix `excludedimports`, more predictable order how hooks are applied #1651
- Internal improvements and code clean-up (#1754, #1760, #1794, #1858, #1862, #1887, #1907, #1913)
- Clean-ups fixes and improvements for the test suite

Known Issues

- Apps built with Windows 10 and Python 3.5 may not run on Windows versions earlier than 10 (#1566).
- The multipackage (MERGE) feature (#1527) is currently broken.
- (OSX) Support for OpenDocument events (#1309) is broken.

3.1.1 (2016-01-31)

Fixed the following issues:

- Fix problems with `setuptools` 19.4 (#1772, #1773, #1790, #1791)
- 3.1 does not collect certain direct imports (#1780)
- Git reports wrong version even if on unchanged release (#1778)
- Don’t resolve symlinks in `modulegraph.py` (#1750, #1755)
- `ShortFileName` not returned in win32 util (#1799)

3.1 (2016-01-09)

- Support reproducible builds (#490, #1434, #1582, #1590).
- Strip leading parts of paths in compiled code objects (#1059, #1302, #1724).
- With `--log-level=DEBUG`, a dependency graph-file is emitted in the build-directory.
- Allow running pyinstaller as user `root`. By popular demand, see e.g. #1564, #1459, #1081.
- New Hooks: `botocore`, `boto3`, `distorm3`, `GObject`, `GI` (G Introspection), `GStreamer`, `GEvent`, `kivy`, `lxml.isoschematron`, `pubsub.core`, `PyQt5.QtMultimedia`, `scipy.linalg`, `shelve`.
- Fixed or Updated Hooks: `astroid`, `django`, `jsonschema` `logilab`, `PyQt4`, `PyQt5`, `skimage`, `sklearn`.
- Add option `--hiddenimport` as an alias for `--hidden-import`.
- (OSX): Fix issues with `st_flags` (#1650).
- (OSX) Remove warning message about 32bit compatibility (#1586).
- (Linux) The cache is now stored in `$XDG_CACHE_HOME/pyinstaller` instead of `$XDG_DATA_HOME` - the cache is moved automatically (#1118).
- Documentation updates, e.g. about reproducible builds
- Put back full text of GPL license into `COPYING.txt`.
- Fix crashes when looking for ctypes DLLs (#1608, #1609, #1620).
- Fix: Imports in byte-code not found if code contains a function (#1581).
- Fix recursion into bytes-code when scanning for ctypes (#1620).
- Fix PyCrypto modules to work with crypto feature (`--key` option) (#1663).
- Fix problems with `excludedimports` in some hook excluding the named modules even if used elsewhere (#1584, #1600).
- Fix freezing of pip 7.1.2 (#1699).
- FreeBSD and Solaris fixes.
- Search for `ldconfig` in `$PATH` first (#1659)
- Deny processing outdated package `_xmlplus`.
- Improvements to the test-suite, testing infrastructure and continuous integration.
- For non-release builds, the exact git revision is not used.
- Internal code refactoring.
- Enhancements and clean-ups to the hooks API - only relevant for hook authors. See the manual for details. E.g:
 - Removed `attrs` in hooks - they were not used anymore anyway.
 - Change `add/del_import()` to accept arbitrary number of module names.
 - New hook utility function `copy_metadata()`.

Known Issues

- Apps built with Windows 10 and Python 3.5 may not run on Windows versions earlier than 10 (#1566).
- The multipackage (MERGE) feature (#1527) is currently broken.
- (OSX) Support for OpenDocument events (#1309) is broken.

3.0 (2015-10-04)

- Python 3 support (3.3 / 3.4 / 3.5).
- Remove support for Python 2.6 and lower.
- Full unicode support in the bootloader (#824, #1224, #1323, #1340, #1396)
 - (Windows) Python 2.7 apps can now run from paths with non-ASCII characters
 - (Windows) Python 2.7 onefile apps can now run for users whose usernames contain non-ASCII characters
 - Fix `sys.getfilesystemencoding()` to return correct values (#446, #885).
- (OSX) Executables built with PyInstaller under OS X can now be digitally signed.
- (OSX) 32bit precompiled bootloader no longer distributed, only 64bit.
- (Windows) for 32bit bootloader enable flag `LARGEADDRESSAWARE` that allows to use 4GB of RAM.
- New hooks: `amazon-product-api`, `appy`, `certifi`, `countrycode`, `cryptography`, `gi`, `httplib2`, `jsonschema`, `keyring`, `lensfunpy`, `mpl_toolkits.basemap`, `ncclient`, `netCDF4`, `OpenCV`, `osgeo`, `patsy`, `PsychoPy`, `pycountry`, `pycparser`, `PyExcelebrate`, `PyGobject`, `pymssql`, `PyNaCl`, `PySiDe.QtCore`, `PySide.QtGui`, `rawpy`, `requests`, `scapy`, `scipy`, `six`, `SpeechRecognition`, `u1db`, `weasyprint`, `Xlib`.
- Hook fixes: `babel`, `ctypes`, `django`, `IPython`, `pint`, `PyEnchant`, `Pygments`, `PyQt5`, `PySide`, `pyusb`, `sphinx`, `sqlalchemy`, `tkinter`, `wxPython`.
- Add support for automatically including data files from eggs.
- Add support for directory eggs support.
- Add support for all kind of namespace packages e.g. `zope.interface`, PEP302 (#502, #615, #665, #1346).
- Add support for `pkgutil.extend_path()`.
- New option `--key` to obfuscate the Python bytecode.
- New option `--exclude-module` to ignore a specific module or package.
- (Windows) New option `--uac-admin` to request admin permissions before starting the app.
- (Windows) New option `--uac-uiaccess` allows an elevated application to work with Remote Desktop.
- (Windows) New options for Side-by-side Assembly searching:
 - `--win-private-assemblies` bundled Shared Assemblies into the application will be changed into Private Assemblies
 - `--win-no-prefer-redirects` while searching for Assemblies PyInstaller will prefer not to follow policies that redirect to newer versions.
- (OSX) New option `--osx-bundle-identifier` to set `.app` bundle identifier.
- (Windows) Remove old COM server support.
- Allow override PyInstaller default config directory by environment variable `PYINSTALLER_CONFIG_DIR`.
- Add FreeBSD support.
- AIX fixes.
- Solaris fixes.
- Use library modulegraph for module dependency analysis.
- Bootloader debug messages `LOADER: . . .` printed to `stderr`.

- PyInstaller no longer extends `sys.path` and bundled 3rd-party libraries do not interfere with their other versions.
- Enhancements to `Analysis()`:
 - New arguments `excludedimports` to exclude Python modules in import hooks.
 - New argument `binaries` to bundle dynamic libraries in `.spec` file and in import hooks.
 - New argument `datas` to bundle additional data files in `.spec` file and in import hooks.
- A lot of internal code refactoring.
- Test suite migrated to `pytest` framework.
- Improved testing infrastructure with continuous integration (Travis - Linux, Appveyor - Windows)
- Wiki and bug tracker migrated to `github`.

Known Issues

- Apps built with Windows 10 and Python 3.5 may not run on Windows versions earlier than 10 (#1566).
- The multipackage (MERGE) feature (#1527) is currently broken.
- (OSX) Support for OpenDocument events (#1309) is broken.

2.1 (2013-09-27)

- Rewritten manual explaining even very basic topics.
- PyInstaller integration with `setuptools` (direct installation with `easy_install` or `pip` from `PYPI` - <https://pypi.python.org/pypi>). After installation there will be available command 'pyinstaller' for PyInstaller usage.
- (Windows) Alter `-version-file` resource format to allow unicode support.
- (Windows) Fix running frozen app running from paths containing foreign characters.
- (Windows) Fix running PyInstaller from paths containing foreign characters.
- (OSX) Implement `-icon` option for the `.app` bundles.
- (OSX) Add `argv` emulation for OpenDocument `AppleEvent` (see manual for details).
- Rename `-buildpath` to `-workpath`.
- Created app is put to `-distpath`.
- All temporary work files are now put to `-workpath`.
- Add option `-clean` to remove PyInstaller cache and temporary files.
- Add experimental support for Linux arm.
- Minimum supported Python version is 2.4.
- Add import hooks for `docutils`, `jinjia2`, `sphinx`, `pytz`, `idlelib`, `sqlite3`.
- Add import hooks for `IPython`, `Scipy`, `pygst`, Python for `.NET`.
- Add import hooks for `PyQt5`, `Bacon`, `raven`.
- Fix `django` import hook to work with Django 1.4.
- Add `rthook` for `twisted`, `pygst`.
- Add `rthook` for `pkg_resource`. It fixes the following functions for frozen app `pkg_resources.resource_stream()`, `pkg_resources.resource_string()`.

- Better support for pkg_resources (.egg manipulation) in frozen executables.
- Add option `--runtime-hook` to allow running custom code from frozen app before loading other Python from the frozen app. This is useful for some specialized preprocessing just for the frozen executable. E.g. this option can be used to set SIP api v2 for PyQt4.
- Fix runtime option `--Wignore`.
- Rename utils to lowercase: `achieve_viewer.py`, `bindepend.py`, `build.py`, `grab_version.py`, `make_comserver.py`, `makespec.py`, `set_version.py`.
- (OSX) Fix missing `qt_menu.nib` in dist directory when using PySide.
- (OSX) Fix bootloader compatibility with Mac OS X 10.5
- (OSX) Search `libpython` in `DYLD_LIBRARY_PATH` if `libpython` cannot be found.
- (OSX) Fix Python library search in `virtualenv`.
- Environment variable `PYTHONHOME` is now unset and path to python home is set in bootloader by function `Py_SetPythonHome()`. This overrides `sys.prefix` and `sys.exec_prefix` for frozen application.
- Python library filename (e.g. `python27.dll`, `libpython2.7.so.1.0`, etc) is embedded to the created exe file. Bootloader is not trying several filenames anymore.
- Frozen executables now use PEP-302 import hooks to import frozen modules and C extensions. (`sys.meta_path`)
- Drop old import machinery from `iu.py`.
- Drop own code to import modules from zip archives (.egg files) in frozen executables. Native Python implementation is kept unchanged.
- Drop old crypto code. This feature was never completed.
- Drop bootloader dependency on Python headers for compilation.
- (Windows) Recompile bootloaders with VS2008 to ensure win2k compatibility.
- (Windows) Use 8.3 filenames for `homepath/temppath`.
- Add prefix `LOADER` to the debug text from bootloader.
- Allow running PyInstaller programmatically.
- Move/Rename some files, code refactoring.
- Add more tests.
- Tilde is in PyInstaller recognized as `$HOME` variable.

2.0 (2012-08-08)

- Minimum supported Python version is 2.3.
- (OSX) Add support for Mac OS X 64-bit
- (OSX) Add support Mac OS X 10.7 (Lion) and 10.8 (Mountain Lion).
- (OSX) With argument `--windowed` PyInstaller creates application bundle (.app)
- automatically.
- Add experimental support for AIX (thanks to Martin Gamwell Dawids).
- Add experimental support for Solaris (thanks to Hywel Richards).
- Add `Multipackage` function to create a collection of packages to avoid

- library duplication. See documentation for more details.
- New simplified command line interface. `Configure.py/Makespec.py/Build.py`
- replaced by `pyinstaller.py`. See documentation for more details.
- Removed cross-building/bundling feature which was never really finished.
- Added option `-log-level` to all scripts to adjust level of output (thanks to Hartmut Goebel).
- `rthooks.dat` moved to `support/rthooks.dat`
- Packaged executable now returns the same return-code as the
- unpackaged script (thanks to Brandyn White).
- Add import hook for PyUSB (thanks to Chien-An “Zero” Cho).
- Add import hook for `wx.lib.pubsub` (thanks to Daniel Hyams).
- Add import hook for `pyttsx`.
- Improve import hook for Tkinter.
- Improve import hook for PyQt4.
- Improve import hook for `win32com`.
- Improve support for running PyInstaller in `virtualenv`.
- Add cli options `-additional-hooks-dir` and `-hidden-import`.
- Remove cli options `-X`, `-K`, `-C`, `-upx`, `-tk`, `-configfile`, `-skip-configure`.
- UPX is used by default if available in the `PATH` variable.
- Remove compatibility code for old platforms (`dos`, `os2`, `MacOS 9`).
- Use Python logging system for message output (thanks to Hartmut Goebel).
- Environment variable `MEIPASS2` is accessible as `sys._MEIPASS`.
- Bootloader now overrides `PYTHONHOME` and `PYTHONPATH`. `PYTHONHOME` and `PYTHONPATH` is set to the value of `MEIPASS2` variable.
- Bootloader uses absolute paths.
- (OSX) Drop dependency on `otool` from Xcode on Mac OSX.
- (OSX) Fix missing `qt_menu.nib` in `dist` directory when using PyQt4.
- (OSX) Bootloader does not use `DYLD_LIBRARY_PATH` on Mac OS X anymore. `@loader_path` is used instead.
- (OSX) Add support to detect `.dylib` dependencies on Mac OS X containing `@executable_path`, `@loader_path` and `@rpath`.
- (OSX) Use `macholib` to detect dependencies on dynamic libraries.
- Improve test suite.
- Improve source code structure.
- Replace `os.system()` calls by `subprocess` module.
- Bundle fake ‘site’ module with frozen applications to prevent loading any user’s Python modules from host OS.
- Include runtime hooks (`rthooks`) in code analysis.
- Source code hosting moved to github: <https://github.com/pyinstaller/pyinstaller>

- Hosting for running tests daily: <https://jenkins.shiningpanda-ci.com/pyinstaller/>

1.5.1 (2011-08-01)

- New default PyInstaller icon for generated executables on Windows.
- Add support for Python built with `--enable-shared` on Mac OSX.
- Add requirements section to documentation.
- Documentation is now generated by `rst2html` and `rst2pdf`.
- Fix wrong path separators for bootloader-file on Windows
- Add workaround for incorrect `platform.system()` on some Python Windows installation where this function returns 'Microsoft' instead 'Windows'.
- Fix `--windowed` option for Mac OSX where a console executable was created every time even with this option.
- Mention dependency on `otool`, `ldd` and `objdump` in documentation.
- Fix typo preventing detection of DLL libraries loaded by `ctypes` module.

1.5 (2011-05-05)

- Full support for Python 2.7.
- Full support for Python 2.6 on Windows. No manual redistribution of DLLs, CRT, manifest, etc. is required: PyInstaller is able to bundle all required dependencies (thanks to Florian Hoech).
- Added support for Windows 64-bit (thanks to Martin Zibricky).
- Added binary bootloaders for Linux (32-bit and 64-bit, using LSB), and Darwin (32-bit). This means that PyInstaller users on this platform don't need to compile the bootloader themselves anymore (thanks to Martin Zibricky and Lorenzo Mancini).
- Rewritten the build system for the bootloader using `waf` (thanks to Martin Zibricky)
- Correctly detect Python unified binary under Mac OSX, and bail out if the unsupported 64-bit version is used (thanks to Nathan Weston).
- Fix TkInter support under Mac OSX (thanks to Lorenzo Mancini).
- Improve bundle creation under Mac OSX and correctly support also one-dir builds within bundles (thanks to Lorenzo Mancini).
- Fix spurious `KeyError` when using `dbhash`
- Fix import of nested packages made from Pyrex-generated files.
- PyInstaller is now able to follow dependencies of binary extensions (`.pyd/.so`) compressed within `.egg`-files.
- Add import hook for `PyTables`.
- Add missing import hook for `QtWebKit`.
- Add import hook for `pywinauto`.
- Add import hook for `reportlab` (thanks Nevar).
- Improve `matplotlib` import hook (for Mac OSX).
- Improve `Django` import hooks.

- Improve compatibility across multiple Linux distributions by being more careful on which libraries are included/excluded in the package.
- Improve compatibility with older Python versions (Python 2.2+).
- Fix double-bouncing-icon bug on Mac OSX. Now windowed applications correctly start on Mac OSX showing a single bouncing icon.
- Fix weird “missing symbol” errors under Mac OSX (thanks to Isaac Wagner).

1.4 (2010-03-22)

- Fully support up to Python 2.6 on Linux/Mac and Python 2.5 on Windows.
- Preliminary Mac OSX support: both one-file and one-dir is supported; for non-console applications, a bundle can be created. Thanks to many people that worked on this across several months (Daniele Zannotti, Matteo Bertini, Lorenzo Mancini).
- Improved Linux support: generated executables are fatter but now should now run on many different Linux distributions (thanks to David Mugnai).
- Add support for specifying data files in import hooks. PyInstaller can now automatically bundle all data files or plugins required for a certain 3rd-party package.
- Add intelligent support for ctypes: PyInstaller is now able to track all places in the source code where ctypes is used and automatically bundle dynamic libraries accessed through ctypes. (Thanks to Lorenzo Mancini for submitting this). This is very useful when using ctypes with custom-made dynamic libraries.
- Executables built with PyInstaller under Windows can now be digitally signed.
- Add support for absolute imports in Python 2.5+ (thanks to Arve Knudsen).
- Add support for relative imports in Python 2.5+.
- Add support for cross-compilation: PyInstaller is now able to build Windows executables when running under Linux. See documentation for more details.
- Add support for .egg files: PyInstaller is now able to look for dependencies within .egg files, bundle them and make them available at runtime with all the standard features (entry-points, etc.).
- Add partial support for .egg directories: PyInstaller will treat them as normal packages and thus it will not bundle metadata.
- Under Linux/Mac, it is now possible to build an executable even when a system packages does not have .pyc or .pyo files available and the system-directory can be written only by root. PyInstaller will in fact generate the required .pyc/.pyo files on-the-fly within a build-temporary directory.
- Add automatic import hooks for many third-party packages, including:
 - PyQt4 (thanks to Pascal Veret), with complete plugin support.
 - pyodbc (thanks to Don Dwiggin)
 - cElementTree (both native version and Python 2.5 version)
 - lxml
 - SQLAlchemy (thanks to Greg Copeland)
 - email in Python 2.5 (though it does not support the old-style Python 2.4 syntax with Python 2.5)
 - gadfly
 - PyQWt5

- mako
- Improved PyGTK (thanks to Marco Bonifazi and foxx).
- paste (thanks to Jamie Kirkpatrick)
- matplotlib
- Add fix for the very annoying “MSVCRT71 could not be extracted” bug, which was caused by the DLL being packaged twice (thanks to Idris Aykun).
- Removed C++-style comments from the bootloader for compatibility with the AIX compiler.
- Fix support for .py files with DOS line endings under Linux (fixes PyOpenGL).
- Fix support for PIL when imported without top-level package (“import Image”).
- Fix PyXML import hook under NT (thanks to Lorenzo Mancini)
- Fixed problem with PyInstaller picking up the wrong copy of optparse.
- Improve correctness of the binary cache of UPX’d/strip’d files. This fixes problems when switching between multiple versions of the same third-party library (like e.g. wxPython allows to do).
- Fix a stupid bug with modules importing optparse (under Linux) (thanks to Louai Al-Khanji).
- Under Python 2.4+, if an exception is raised while importing a module inside a package, the module is now removed from the parent’s namespace (to match the behaviour of Python itself).
- Fix random race-condition at startup of one-file packages, that was causing this exception to be generated: “PYZ entry ‘encodings’ (0j) is not a valid code object”.
- Fix problem when having unicode strings among path elements.
- Fix random exception (“bad file descriptor”) with “prints” in non-console mode (actually a pythonw “bug” that’s fixed in Python 3.0).
- Sometimes the temporary directory did not get removed upon program exit, when running on Linux.
- Fixed random segfaults at startup on 64-bit platforms (like x86-64).

1.3 (2006-12-20)

- Fix bug with user-provided icons disappearing from built executables when these were compressed with UPX.
- Fix problems with packaging of applications using PIL (that was broken because of a bug in Python’s import machinery, in recent Python versions). Also add a workaround including Tcl/Tk with PIL unless ImageTk is imported.
- (Windows) When used under Windows XP, packaged programs now have the correct look & feel and follow user’s themes (thanks to the manifest file being linked within the generated executable). This is especially useful for applications using wxPython.
- Fix a buffer overrun in the bootloader (which could lead to a crash) when the built executable is run from within a deep directory (more than 70-80 characters in the pathname).
- Bootstrap modules are now compressed in the executable (so that they are not visible in plaintext by just looking at it with a hex editor).
- Fixed a regression introduced in 1.1: under Linux, the bootloader does not depend on libpythonX.X.so anymore.

1.2 (2006-06-29)

- Fix a crash when invoking UPX with certain kinds of builds.
- Fix icon support by re-adding a resource section in the bootloader executable.

1.1 (2006-02-13)

- (Windows) Make single-file packages not depend on MSVCRT71.DLL anymore, even under Python 2.4. You can eventually ship your programs really as single-file executables, even when using the newest Python version!
- Fix problem with incorrect python path detection. Now using helpers from distutils.
- Fix problem with rare encodings introduced in newer Python versions: now all the encodings are automatically found and included, so this problem should be gone forever.
- Fix building of COM servers (was broken in 1.0 because of the new build system).
- Mimic Python 2.4 behaviour with broken imports: sys.modules is cleaned up afterwise. This allows to package SQLAlchemy applications under Windows with Python 2.4 and above.
- Add import hook for the following packages:
 - GTK
 - PyOpenGL (tested 2.0.1.09)
 - dsnpython (tested 1.3.4)
 - KInterasDB (courtesy of Eugene Prigorodov)
- Fix packaging of code using “time.strptime” under Python 2.3+.
- (Linux) Ignore linux-gate.so while calculating dependencies (fix provided by Vikram Aggarwal).
- (Windows) With Python 2.4, setup UPX properly so to be able to compress binaries generated with Visual Studio .NET 2003 (such as most of the extensions). UPX 1.92+ is needed for this.

1.0 (2005-09-19) with respect to McMillan’s Python Installer 5b5

- Add support for Python 2.3 (fix packaging of codecs).
- Add support for Python 2.4 (under Windows, needed to recompile the bootloader with a different compiler version).
- Fix support for Python 1.5.2, should be fully functional now (required to rewrite some parts of the string module for the bootloader).
- Fix a rare bug in extracting the dependencies of a DLL (bug in PE header parser).
- Fix packaging of PyQt programs (needed an import hook for a hidden import).
- Fix imports calculation for modules using the “from __init__ import” syntax.
- Fix a packaging bug when a module was being import both through binary dependency and direct import.
- Restyle documentation (now using docutils and reStructuredText).
- New Windows build system for automatic compilations of bootloader in all the required flavours (using Scons)

Credits

Thanks goes to all the kind PyInstaller contributors who have contributed new code, bug reports, fixes, comments and ideas. A brief list follows, please let us know if your name is omitted by accident:

Contributions to PyInstaller 3.2.1

Special Thanks to Thomas Waldmann and David Vierra for support when working on the new build system.

- Hartmut Goebel - Core developer and release manager.
- Martin Zibricky - Core developer.
- David Cortesi - Core developer and documentation manager.
- Bryan A. Jones - Core developer.
- David Vierra - Core developer and encoding specialist.
- Cecil Curry - brave bug-fixing and code-refactoring
- Amane Suzuki
- Andy Cycle
- Axel Huebl
- Bruno Oliveira
- Dan Auerbach
- Daniel Hyams
- Denis Akhyyarov
- Dror Asaf
- Dustin Spicuzza
- Emanuele Bertoldi
- Glenn Ramsey
- Hugh Dowling
- Jesse Suen
- Jonathan Dan
- Jonathan Springer
- Jonathan Stewmon
- Julie Marchant
- Kenneth Zhao
- Linus Groh
- Mansour Moufid
- Martin Zibricky
- Matteo Bertini
- Nicolas Dickreuter

- Peter Würtz
- Ronald Oussoren
- Santiago Reig
- Sean Fisk
- Sergei Litvinchuk
- Stephen Rauch
- Thomas Waldmann
- Till Bald
- xoviat

Contributions to PyInstaller 3.2

- Hartmut Goebel - Core developer and release manager.
- Martin Zibricky - Core developer.
- David Cortesi - Core developer and documentation manager.
- Bryan A. Jones - Core developer.
- David Vierra - Core developer and encoding specialist.
- Cecil Curry - brave bug-fixing and code-refactoring
- And Cycle - unicode fixes.
- Chris Hager - QtQuick hook.
- David Schooris - wrong icon parameter in Windows example.
- Florian Bruhin - typo hunting.
- Garth Bushell - Support for objcopy.
- Insoleet - lib2to3 hook
- Jonathan Springer - hook fixes, brave works on PyQt.
- Matteo Bertini - code refactoring.
- Jonathan Stewmon - bug hunting.
- Kenneth Zhao - waf update.
- Leonid Rozenberg - typo hunting.
- Merlijn Wajer - bug fixing.
- Nicholas Chammas - cleanups.
- nih - hook fixes.
- Olli-Pekka Heinisuo - CherryPy hook.
- Rui Carmo - cygwin fixes.
- Stephen Rauch - hooks and fixes for unnecessary rebuilds.
- Tim Stumbaugh - bug hunting.

Contributions to PyInstaller 3.1.1

- Hartmut Goebel - Core developer and release manager.
- David Vierra - Core developer and encoding specialist.
- Torsten Landschoff - Fix problems with setuptools
- Peter Inglesby - resolve symlinks in modulegraph.py
- syradium - bug hunting
- dessant - bug hunting
- Joker Qyou - bug hunting

Contributions to PyInstaller 3.1

- Hartmut Goebel - Core developer and release manager.
- Martin Zibricky - Core developer.
- David Cortesi - Core developer and documentation manager.
- Bryan A. Jones - Core developer.
- David Vierra - Core developer and encoding specialist.
- Andrei Kopats - Windows fixes.
- Andrey Malkov - Django runtime hooks.
- Ben Hagen - kivy hook, GStreamer realtime hook.
- Cecil Curry - Module Version Comparisons and reworking hooks.
- Dustin Spicuzza - Hooks for GLib, GIntrospection, Gstreamer, etc.
- giumas - lxml.isoschematron hook.
- Jonathan Stewmon - Hooks for botocore, boto, boto3 and gevent.monkey.
- Kenneth Zhao - Solaris fixes.
- Matthew Einhorn - kivy hook.
- mementum - pubsub.core hook.
- Nicholas Chammas - Documentation updates.
- Nico Galoppo - Hooks for skimage and sklearn.
- Panagiotis H.M. Issaris - weasyprint hook.
- Penaz - shelve hook.
- Roman Yurchak - scipy.linalg hook.
- Starwarsfan2099 - Distorm3 hook.
- Thomas Waldmann - Fixes for Bootloader and FreeBSD.
- Tim Stumbaugh - Bug fixes.
- zpin - Bug fixes.

Contributions to PyInstaller 3.0

- Martin Zibricky - Core developer and release manager.
- Hartmut Goebel - Core developer.
- David Cortesi - Initial work on Python 3 support, Python 3 fixes, documentation updates, various hook fixes.
- Cecil Curry - 'six' hook for Python 3, various modulegraph improvements, wxPython hook fixes,
- David Vierra - unicode support in bootloader, Windows SxS Assembly Manifest fixes and many other Windows improvements.
- Michael Mulley - keyring, PyNaCl import hook.
- Rainer Dreyer - OS X fixes, hook fixes.
- Bryan A. Jones - test suite fixes, various hook fixes.
- Philippe Pepiot - Linux fixes.
- Emanuele Bertoldi - pycountry import hook, Django import hook fixes.
- Glenn Ramsey - PyQt5 import hook - support for QtWebEngine on OSX, various hook fixes, Windows fixes.
- Karol Woźniak - import hook fixes.
- Jonathan Springer - PyGObject hooks. ctypes, PyEnchant hook fixes, OS X fixes.
- Giuseppe Masetti - osgeo, mpl_toolkits.basemap and netCDF4 import hooks.
- Yuu Yamashita - OS X fixes.
- Thomas Waldmann - FreeBSD fixes.
- Boris Savelev - FreeBSD and Solaris fixes.
- Guillermo Gutiérrez - Python 3 fixes.
- Jasper Geurtz - gui fixes, hook fixes.
- Holger Pandel - Windows fixes.
- Anthony Zhang - SpeechRecognition import hook.
- Andrei Fokau - Python 3.5 fixes.
- Kenneth Zhao - AIX fixes.
- Maik Riechert - lensfunpy, rawpy import hooks.
- Tim Stumbaugh - hook fixes.
- Andrew Leech - Windows fixes.
- Patrick Robertson - tkinter import hook fixes.
- Yaron de Leeuw - import hook fixes.
- Bryan Cort - PsychoPy import hook.
- Phoebus Veiz - bootloader fixes.
- Sean Johnston - version fix.
- Kevin Zhang - PyExceerate import hook.
- Paulo Matias - unicode fixes.
- Lorenzo Villani - crypto feature, various fixes.

- Janusz Skonieczny - hook fixes.
- Martin Gamwell Dawids - Solaris fixes.
- Volodymyr Vitvitskyi - typo fixes.
- Thomas Kho - django import hook fixes.
- Konstantinos Koukopoulos - FreeBSD support.
- Jonathan Beezley - PyQt5 import hook fixes.
- Andraz Vrhovc - various fixes.
- Noah Treuhaft - OpenCV import hook.
- Michael Hipp - reportlab import hook.
- Michael Sverdlik - certifi, httplib2, requests, jsonschema import hooks.
- Santiago Reig - appy import hook.

Contributions to PyInstaller 2.1 and older

- Glenn Ramsey - PyQt5 import hook.
- David Cortesi - PyInstaller manual rewrite.
- Vaclav Smilauer - IPython import hook.
- Shane Hansen - Linux arm support.
- Bryan A. Jones - docutils, jinja2, sphinx, pytz, idlelib import hooks.
- Patrick Stewart <patstew at gmail dot com> - scipy import hook.
- Georg Schoelly <mail at georg-schoelly dot com> - storm ORM import hook.
- Vinay Sajip - zmq import hook.
- Martin Gamwell Dawids - AIX support.
- Hywel Richards - Solaris support.
- Brandyn White - packaged executable return code fix.
- Chien-An “Zero” Cho - PyUSB import hook.
- Daniel Hyams - h2py, wx.lib.pubsub import hooks.
- Hartmut Goebel - Python logging system for message output. Option `-log-level`.
- Florian Hoeh - full Python 2.6 support on Windows including automatic handling of DLLs, CRT, manifest, etc. Read and write resources from/to Win32 PE files.
- Martin Zibricky - rewrite the build system for the bootloader using waf. LSB compliant precompiled bootloaders for Linux. Windows 64-bit support.
- Peter Burgers - matplotlib import hook.
- Nathan Weston - Python architecture detection on OS X.
- Isaac Wagner - various OS X fixes.
- Matteo Bertini - OS X support.
- Daniele Zannotti - OS X support.
- David Mugnai - Linux support improvements.

- Arve Knudsen - absolute imports in Python 2.5+
- Pascal Veret - PyQt4 import hook with Qt4 plugins.
- Don Dwigins - pyodbc import hook.
- Allan Green - refactoring and improved in-process COM servers.
- Daniele Varrazzo - various bootloader and OS X fixes.
- Greg Copeland - sqlalchemy import hook.
- Seth Remington - PyGTK hook improvements.
- Marco Bonifazi - PyGTK hook improvements. PyOpenGL import hook.
- Jamie Kirkpatrick - paste import hook.
- Lorenzo Mancini - PyXML import hook fixes under Windows. OS X support. App bundle creation on OS X. Tkinter on OS X. Precompiled bootloaders for OS X.
- Lorenzo Berni - django import hook.
- Louai Al-Khanji - fixes with optparse module.
- Thomas Heller - set custom icon of Windows exe files.
- Eugene Prigorodov <eprigorodov at naumen dot ru> - KInterasDB import hook.
- David C. Morrill - vtkpython import hook.
- Alan James Salmoni - Tkinter interface to PyInstaller.

Man Pages

pyinstaller

SYNOPSIS

```
pyinstaller <options> SCRIPT
```

DESCRIPTION

Automatically calls pyi-configure, pyi-makespec and pyi-build in one run. In most cases, running `pyinstaller` will be all you have to do.

Please see the PyInstaller Manual for more information.

OPTIONS

- | | |
|----------------------------|--|
| -h, --help | show this help message and exit |
| -v, --version | Show program version info and exit. |
| --distpath DIR | Where to put the bundled app (default: <code>./dist</code>) |
| --workpath WORKPATH | Where to put all the temporary work files, <code>.log</code> , <code>.pyz</code> and etc. (default: <code>./build</code>) |

- y, --noconfirm** Replace output directory (default: SPEC_PATH/dist/SPEC_NAME) without asking for confirmation
- upx-dir UPX_DIR** Path to UPX utility (default: search the execution path)
- a, --ascii** Do not include unicode encoding support (default: included if available)
- clean** Clean PyInstaller cache and remove temporary files before building.
- log-level LEVEL** Amount of detail in build-time console messages. LEVEL may be one of TRACE, DEBUG, INFO, WARN, ERROR, CRITICAL (default: INFO).

What to generate

- D, --onedir** Create a one-folder bundle containing an executable (default)
- F, --onefile** Create a one-file bundled executable.
- specpath DIR** Folder to store the generated spec file (default: current directory)
- n NAME, --name NAME** Name to assign to the bundled app and spec file (default: first script's basename)

What to bundle, where to search

- add-data <SRC;DEST or SRC:DEST>** Additional non-binary files or folders to be added to the executable. The path separator is platform specific, `os.pathsep` (which is `;` on Windows and `:` on most unix systems) is used. This option can be used multiple times.
- add-binary <SRC;DEST or SRC:DEST>** Additional binary files to be added to the executable. See the `--add-data` option for more details. This option can be used multiple times.
- p DIR, --paths DIR** A path to search for imports (like using PYTHONPATH). Multiple paths are allowed, separated by `:`, or use this option multiple times
- hidden-import MODULENAME, --hiddenimport MODULENAME** Name an import not visible in the code of the script(s). This option can be used multiple times.
- additional-hooks-dir HOOKSPATH** An additional path to search for hooks. This option can be used multiple times.
- runtime-hook RUNTIME_HOOKS** Path to a custom runtime hook file. A runtime hook is code that is bundled with the executable and is executed before any other code or module to set up special features of the runtime environment. This option can be used multiple times.
- exclude-module EXCLUDES** Optional module or package (the Python name, not the path name) that will be ignored (as though it was not found). This option can be used multiple times.
- key KEY** The key used to encrypt Python bytecode.

How to generate

- d, --debug** Tell the bootloader to issue progress messages while initializing and starting the bundled app. Used to diagnose problems with missing imports.

- s, --strip** Apply a symbol-table strip to the executable and shared libs (not recommended for Windows)
- noupX** Do not use UPX even if it is available (works differently between Windows and *nix)

Windows and Mac OS X specific options

- c, --console, --nowindowed** Open a console window for standard i/o (default)
- w, --windowed, --noconsole** Windows and Mac OS X: do not provide a console window for standard i/o. On Mac OS X this also triggers building an OS X .app bundle. This option is ignored in *NIX systems.
- i <FILE.ico or FILE.exe,ID or FILE.icns>, --icon <FILE.ico or FILE.exe,ID or FILE.icns>**
FILE.ico: apply that icon to a Windows executable. FILE.exe,ID, extract the icon with ID from an exe. FILE.icns: apply the icon to the .app bundle on Mac OS X

Windows specific options

- version-file FILE** add a version resource from FILE to the exe
- m <FILE or XML>, --manifest <FILE or XML>** add manifest FILE or XML to the exe
- r RESOURCE, --resource RESOURCE** Add or update a resource to a Windows executable. The RESOURCE is one to four items, FILE[,TYPE[,NAME[,LANGUAGE]]]. FILE can be a data file or an exe/dll. For data files, at least TYPE and NAME must be specified. LANGUAGE defaults to 0 or may be specified as wildcard * to update all resources of the given TYPE and NAME. For exe/dll files, all resources from FILE will be added/updated to the final executable if TYPE, NAME and LANGUAGE are omitted or specified as wildcard *. This option can be used multiple times.
- uac-admin** Using this option creates a Manifest which will request elevation upon application restart.
- uac-uiaccess** Using this option allows an elevated application to work with Remote Desktop.

Windows Side-by-side Assembly searching options (advanced)

- win-private-assemblies** Any Shared Assemblies bundled into the application will be changed into Private Assemblies. This means the exact versions of these assemblies will always be used, and any newer versions installed on user machines at the system level will be ignored.
- win-no-prefer-redirects** While searching for Shared or Private Assemblies to bundle into the application, PyInstaller will prefer not to follow policies that redirect to newer versions, and will try to bundle the exact versions of the assembly.

Mac OS X specific options

- osx-bundle-identifier BUNDLE_IDENTIFIER** Mac OS X .app bundle identifier is used as the default unique program name for code signing purposes. The usual

form is a hierarchical name in reverse DNS notation. For example:
com.mycompany.department.appname (default: first script's basename)

ENVIRONMENT VARIABLES

| | |
|-----------------------------|--|
| PYIN- STALLER_CONFIG_DIR | This changes the directory where PyInstaller caches some files. The default location for this is OS operating system dependent, but is typically a subdirectory of the home directory. |
|-----------------------------|--|

SEE ALSO

`pyi-configure(1)`, `pyi-makespec(1)`, `pyi-build(1)`, The PyInstaller Manual, `pyinstaller(1)`

Project Homepage <http://www.pyinstaller.org>

pyi-makespec

SYNOPSIS

```
pyi-makespec <options> SCRIPT [SCRIPT ...]
```

DESCRIPTION

The spec file is the description of what you want *PyInstaller* to do with your program. `pyi-makespec` is a simple wizard to create spec files that cover basic usages:

```
pyi-makespec [--onefile] yourprogram.py
```

By default, `pyi-makespec` generates a spec file that tells *PyInstaller* to create a distribution directory contains the main executable and the dynamic libraries. The option `--onefile` specifies that you want PyInstaller to build a single file with everything inside.

In most cases the specfile generated by `pyi-makespec` is all you need. If not, see *When things go wrong* in the manual and be sure to read the introduction to *Spec Files*.

OPTIONS

- h, --help** show this help message and exit
- log-level LEVEL** Amount of detail in build-time console messages. LEVEL may be one of TRACE, DEBUG, INFO, WARN, ERROR, CRITICAL (default: INFO).

What to generate

- D, --onedir** Create a one-folder bundle containing an executable (default)
- F, --onefile** Create a one-file bundled executable.
- specpath DIR** Folder to store the generated spec file (default: current directory)
- n NAME, --name NAME** Name to assign to the bundled app and spec file (default: first script's basename)

What to bundle, where to search

- add-data <SRC;DEST or SRC:DEST>** Additional non-binary files or folders to be added to the executable. The path separator is platform specific, `os.pathsep` (which is `;` on Windows and `:` on most unix systems) is used. This option can be used multiple times.
- add-binary <SRC;DEST or SRC:DEST>** Additional binary files to be added to the executable. See the `--add-data` option for more details. This option can be used multiple times.
- p DIR, --paths DIR** A path to search for imports (like using `PYTHONPATH`). Multiple paths are allowed, separated by `';`, or use this option multiple times
- hidden-import MODULENAME, --hiddenimport MODULENAME** Name an import not visible in the code of the script(s). This option can be used multiple times.
- additional-hooks-dir HOOKSPATH** An additional path to search for hooks. This option can be used multiple times.
- runtime-hook RUNTIME_HOOKS** Path to a custom runtime hook file. A runtime hook is code that is bundled with the executable and is executed before any other code or module to set up special features of the runtime environment. This option can be used multiple times.
- exclude-module EXCLUDES** Optional module or package (the Python name, not the path name) that will be ignored (as though it was not found). This option can be used multiple times.
- key KEY** The key used to encrypt Python bytecode.

How to generate

- d, --debug** Tell the bootloader to issue progress messages while initializing and starting the bundled app. Used to diagnose problems with missing imports.
- s, --strip** Apply a symbol-table strip to the executable and shared libs (not recommended for Windows)
- noupx** Do not use UPX even if it is available (works differently between Windows and *nix)

Windows and Mac OS X specific options

- c, --console, --nowindowed** Open a console window for standard i/o (default)
- w, --windowed, --noconsole** Windows and Mac OS X: do not provide a console window for standard i/o. On Mac OS X this also triggers building an OS X `.app` bundle. This option is ignored in *NIX systems.
- i <FILE.ico or FILE.exe,ID or FILE.icns>, --icon <FILE.ico or FILE.exe,ID or FILE.icns>**
FILE.ico: apply that icon to a Windows executable. FILE.exe,ID, extract the icon with ID from an exe. FILE.icns: apply the icon to the `.app` bundle on Mac OS X

Windows specific options

- version-file FILE** add a version resource from FILE to the exe
- m <FILE or XML>, --manifest <FILE or XML>** add manifest FILE or XML to the exe
- r RESOURCE, --resource RESOURCE** Add or update a resource to a Windows executable. The RESOURCE is one to four items, FILE[,TYPE[,NAME[,LANGUAGE]]]. FILE can be a data file or an exe/dll. For data files, at least TYPE and NAME must be specified. LANGUAGE defaults to 0 or may be specified as wildcard * to update all resources of the given TYPE and NAME. For exe/dll files, all resources from FILE will be added/updated to the final executable if TYPE, NAME and LANGUAGE are omitted or specified as wildcard *. This option can be used multiple times.
- uac-admin** Using this option creates a Manifest which will request elevation upon application restart.
- uac-uiaccess** Using this option allows an elevated application to work with Remote Desktop.

Windows Side-by-side Assembly searching options (advanced)

- win-private-assemblies** Any Shared Assemblies bundled into the application will be changed into Private Assemblies. This means the exact versions of these assemblies will always be used, and any newer versions installed on user machines at the system level will be ignored.
- win-no-prefer-redirects** While searching for Shared or Private Assemblies to bundle into the application, PyInstaller will prefer not to follow policies that redirect to newer versions, and will try to bundle the exact versions of the assembly.

Mac OS X specific options

- osx-bundle-identifier BUNDLE_IDENTIFIER** Mac OS X .app bundle identifier is used as the default unique program name for code signing purposes. The usual form is a hierarchical name in reverse DNS notation. For example: com.mycompany.department.appname (default: first script's basename)

ENVIRONMENT VARIABLES

| | |
|----------------------------|--|
| PYIN- STALLER_CONFIGDIR | This changes the directory where PyInstaller caches some files. The default location for this is OS operating system dependent, but is typically a subdirectory of the home directory. |
|----------------------------|--|

SEE ALSO

pyi-build(1), The PyInstaller Manual, pyinstaller(1)

Project Homepage <http://www.pyinstaller.org>

Indices and tables

- [genindex](#)
- [modindex](#)
- [search](#)

P

Python Enhancement Proposals

PEP 302, 31