## Contents

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 PyInstaller 5  
1.5 What PyInstaller Does and How It Does It 7  
1.6 Using PyInstaller 11  
1.7 Run-time Information 21  
1.8 Using Spec Files 23  
1.9 Notes about specific Features 31  
1.10 When Things Go Wrong 32  
1.11 Advanced Topics 36  
1.12 Understanding PyInstaller Hooks 45  
1.13 Building the Bootloader 52  
1.14 Changelog for PyInstaller 58  
1.15 Credits 83  
1.16 Man Pages 90  
1.17 Development Guide 97  
1.18 Indices and tables 106

Index 107
PyInstaller Documentation, Release 3.5

Version PyInstaller 3.5

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 3.4+, and correctly bundles the major Python packages such as numpy, PyQt, Django, wxPython, and others.

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

1.1 Requirements

1.1.1 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.

1.1.2 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.

1.1.3 GNU/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`.

### 1.1.4 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*.

### 1.2 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.

### 1.3 How To Contribute

You are very welcome to contribute! *PyInstaller* is a maintained by a group of volunteers. All contributions, like community support, bug reports, bug fixes, documentation improvements, enhancements and ideas are welcome.

*PyInstaller* is an free software project that is created and maintained by volunteers. It lives-and-dies based on the support it receives from others, and the fact that you’re even considering contributing to *PyInstaller* is very generous of you.

Since as of now all core-developers are working on PyInstaller in their spare-time, you can help us (and the project) most if you are following some simple guidelines. The higher the quality of your contribution, the less work we have incorporating them and the earlier we will be able to incorporate them :-)

If you get stuck at any point you can ask on the PyInstaller Email List or create a ticket on GitHub.

For more about our development process and methods, see the Development Guide.

### 1.3.1 Some ideas how you can help

Some ideas how you can help:

- **Subscribe** to the mailing list (low traffic) or **join** the IRC channel and share your experience or answer questions from the community.
- **Answer support tickets**: Often the user just needs to be pointed to the fitting section in the manual.

- **Triage open issues**, which means: read the report; ask the issue requester to provide missing information and to try with the latest development version; ensure there is a *minimal* example; ensure the issue-reporter followed all steps in *When Things Go Wrong*. If you are able reproduce the problem and track down the bug, this would be a *great* help for the core developers.

- **Help improving the documentation**: There is a list of documentation issues you can pick one from. Please provide a pull-request for your changes. *Read more »*

- **Pick an issue requesting a pull-request** and provide one.

- **Review pull requests**: Are the commit messages following the guideline *Please Write Good Commit Messages*; do all new files have a copyright-header (esp. for hooks this is often missing); is the code okay; etc.

- **Scan the list of open issues** and pick some task :-)

Thank you very much!

If you plan to contribute frequently, just ask for write access to the main git repository. We would be glad to welcome you in the team!

**Sponsorship and Project Grant**

Please consider sponsoring *PyInstaller* development, especially if your company benefits from this project.

We welcome your patronage on Bountysource:

- Contribute a recurring amount to the team
- Place a bounty on a specific feature

Your contribution will go towards adding new features to *PyInstaller* and making sure all functionality continues to meet our high quality standards.

A grant for contiguous full-time development has the biggest impact for progress. Periods of 3 to 10 days allow a contributor to tackle substantial complex issues which are otherwise left to linger until somebody can’t afford to not fix them.

Contact Hartmut Goebel to arrange a grant for a core contributor.

Huge thanks to all the companies and individuals who financially contributed to the development of *PyInstaller*. Please send a PR if you’ve donated and would like to be listed on the web-site.

### 1.4 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 is available, for example:

```
$ pip install pyinstaller
```

or upgrade to a newer version:

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

To install the current development version use:
1.4.1 Installing in Windows

For Windows, PyWin32 or the more recent pypiwin32, is a prerequisite. The latter is installed automatically when you install PyInstaller using pip or easy_install. If necessary, follow the pypiwin32 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.

1.4.2 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.

1.4.3 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, GNU/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.
1.4.4 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
- **GNU/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`.

1.4.5 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.)

1.5 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,
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.

### 1.5.1 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, using `imp.find_module()`, 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 GNU/Linux, it does not bundle any file from `/lib` or `/usr/lib`, assuming these will be found in every system.
1.5.2 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 myscript 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 myscript 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.

1.5.3 How the One-Folder Program Works

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

The PyInstaller bootloader is a binary executable program for the active platform (Windows, GNU/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 myscript 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.)

1.5.4 Bundling to One File

PyInstaller can bundle your script and all its dependencies into a single executable named myscript (myscript.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 is much easier to diagnose problems in one-folder mode.

1.5.5 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 xxxxx 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 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 GNU/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. If you know the target environment, --runtime-tmpdir might be a workaround.)

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.

It is possible to control the location of the _MEIxxxxxx folder by using the --runtime-tmpdir command line option. The specified path is stored in the executable, and the bootloader will create the _MEIxxxxxx folder inside of the specified folder. Please see Defining the Extraction Location for details.

Applications that use os.setuid() may encounter permissions errors. The temporary folder where the bundled app runs may not being 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.

1.5.6 Using a Console Window

By default the bootloader creates a command-line console (a terminal window in GNU/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.

As noted in the CPython tutorial Appendix, for Windows a file extention of .pyw suppresses the console window that normally appears. Likewise, a console window will not be provided when using a myscript.pyw script with PyInstaller.

1.5.7 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. See Encrypting Python Bytecode for details.

1.6 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
```

The `myscript.spec` file contains most of the information provided by the options that were specified when `pyinstaller` (or `pyi-makespec`) was run with the script file as the argument. You typically do not need to specify any options when running `pyinstaller` with the spec file. Only a few command-line options have an effect when building from a spec file.

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"
```

1.6.1 Options

General Options

- `-h, --help` show this help message and exit
PyInstaller Documentation, Release 3.5

-\v, --version
Show program version info and exit.

--distpath \DIR
Where to put the bundled app (default: ./dist)

--workpath \WORKPATH
Where to put all the temporary work files, .log, .pyz and etc. (default: ./build)

-y, --noconfirm
Replace output directory (default: SPECPATH/dist/SPECNAME) 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 mod-
ule 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 <all,imports,bootloader,noarchive>, --debug <all,imports,bootloader,noarchive>  Provide assistance with debugging a frozen application. This argument may be provided multiple times to select several of the following options.

- all: All three of the following options.
- imports: specify the -v option to the underlying Python interpreter, causing it to print a message each time a module is initialized, showing the place (filename or built-in module) from which it is loaded. See https://docs.python.org/3/using/cmdline.html#id4.
- bootloader: tell the bootloader to issue progress messages while initializing and starting the bundled app. Used to diagnose problems with missing imports.
- noarchive: instead of storing all frozen Python source files as an archive inside the resulting executable, store them as files in the resulting output directory.

-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)

--upx-exclude FILE  Prevent a binary from being compressed when using upx. This is typically used if upx corrupts certain binaries during compression. FILE is the filename of the binary without path. This option can be used multiple times.

Windows and Mac OS X specific options

-c, --console, --nowindowed  Open a console window for standard i/o (default). On Windows this option will have no effect if the first script is a `.pyw` file.

-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. On Windows this option will be set if the first script is a `.pyw` file. 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)

Rarely used special options

--runtime-tmpdir PATH
Where to extract libraries and support files in onefile-mode. If this option is given, the bootloader will ignore any temp-folder location defined by the runtime OS. The _MEIxxxxxx-folder will be created here. Please use this option only if you know what you are doing.

--bootloader-ignore-signals
Tell the bootloader to ignore signals rather than forwarding them to the child process. Useful in situations where e.g. a supervisor process signals both the bootloader and child (e.g. via a process group) to avoid signalling the child twice.

1.6.2 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 GNU/Linux:

```
pyinstaller --noconfirm --log-level=WARN \
    --onefile --nowindow \ 
    --add-data="README:." \ 
    --add-data="imagel.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:

```bash
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="MLNMFLCN.ICO ^
   myscript.spec
```

### 1.6.3 Running PyInstaller from Python code

If you want to run PyInstaller from within Python code use the `run` function of the `__main__` module and pass all command line arguments in as a list, e.g.

```python
import PyInstaller.__main__

PyInstaller.__main__.run([
    '--name=%s' % package_name,
    '--onefile',
    '--windowed',
    '--add-binary=%s' % os.path.join('resource', 'path', '*.png'),
    '--add-data=%s' % os.path.join('resource', 'path', '*.txt'),
    '--icon=%s' % os.path.join('resource', 'path', 'icon.ico'),
    os.path.join('my_package', '__main__.py'),
])
```

### 1.6.4 Running PyInstaller with Python optimizations

**Note:** When using this feature, you should be aware of how the Python bytecode optimization mechanism works. When using `-O`, `__debug__` is set to `False` and `assert` statements are removed from the bytecode. The `-OO` flag additionally removes docstrings.

Using this feature affects not only your main script, but *all* modules included by PyInstaller. If your code (or any module imported by your script) relies on these features, your program may break or have unexpected behavior.

*PyInstaller* can be run with Python optimization flags (`-O` or `-OO`) by executing it as a Python module, rather than using the `pyinstaller` command:

```bash
# run with basic optimizations
python -O -m PyInstaller myscript.py

# also discard docstrings
python -OO -m PyInstaller myscript.py
```

Or, by explicitly setting the `PYTHONOPTIMIZE` environment variable to a non-zero value:

```bash
# Unix
PYTHONOPTIMIZE=1 pyinstaller myscript.py
```

(continues on next page)
You can use any PyInstaller options that are otherwise available with the pyinstaller command. For example:

```
python -O -m PyInstaller --onefile myscript.py
```

Alternatively, you can also use the path to pyinstaller:

```
python -O /path/to/pyinstaller myscript.py
```

### 1.6.5 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](http://upx.sourceforge.net) 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 starts 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.

### 1.6.6 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.

### 1.6.7 Defining the Extraction Location

In rare cases, when you bundle to a single executable (see Bundling to One File and How the One-File Program Works), you may want to control the location of the temporary directory at compile time. This can be done using the `--runtime-tmpdir` option. If this option is given, the bootloader will ignore any temp-folder location defined by the run-time OS. Please use this option only if you know what you are doing.

### 1.6.8 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, python3 -m venv does the same job, see module venv.)

- 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 GNU/Linux and Mac OS, you switch environments at the command line. See PEP 405 for more information about Python virtual environments.

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 # GNU/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 GNU/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 GNU/Linux using the free Wine environment. Further details are needed, see How to Contribute.
1.6.9 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:

```bash
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:

```plaintext
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 $0002000000040000$ 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:

```bash
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.

1.6.10 Building Mac OS X App Bundles

Under Mac OS X, PyInstaller always builds a UNIX executable in dist. If you specify --onedir, the output is a folder named myscript containing supporting files and an executable named myscript. If you specify --onefile, the output is a single UNIX executable named myscript. Either executable can be started from a Terminal command line. Standard input and output work as normal through that Terminal window.

If you specify --windowed with either option, the dist folder also contains 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` which contains:

- A folder `Frameworks` which is empty.
- A folder `Resources` that contains an icon file.
- A file `Info.plist` that describes the app.
- A folder `MacOS` that contains the executable and supporting files, just as in the `--onedir` folder.

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

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).

You can add other items to the `Info.plist` by editing the spec file; see *Spec File Options for a Mac OS X Bundle* below.

### 1.6.11 Platform-specific Notes

#### GNU/Linux

**Making GNU/Linux Apps Forward-Compatible**

Under GNU/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 GNU/Linux, it may fail to execute (typically with a runtime dynamic link error) if it is executed on an older version of GNU/Linux.

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

The GNU/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.

#### Windows

For Python >= 3.5 targeting *Windows < 10*, the developer needs to take special care to include the Visual C++ run-time .dlls: Python 3.5 uses Visual Studio 2015 run-time, which has been renamed into “Universal CRT” and has become part of Windows 10. For Windows Vista through Windows 8.1 there are Windows Update packages, which may or may not be installed in the target-system. So you have the following options:

1. Build on *Windows 7* which has been reported to work.
2. Include one of the VCRedist packages (the redistributable package files) into your application’s installer. This is Microsoft’s recommended way, see “Distributing Software that uses the Universal CRT“ in the above-mentioned link, numbers 2 and 3.
3. Install the Windows Software Development Kit (SDK) for Windows 10 and expand the `.spec`-file to include the required DLLs, see “Distributing Software that uses the Universal CRT“ in the above-mentioned link, number 6.
If you think, PyInstaller should do this by itself, please help improving PyInstaller.

Mac OS X

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 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:

```bash
$ 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:

```bash
$ /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:

```bash
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.

### 1.7 Run-time Information

Your app should run in a bundle exactly as it does when run from source. However, you may want to learn at run-time whether the app is running from source or whether it is bundled (“frozen”). You can use the following code to check “are we bundled?”:

```python
import sys
if getattr(sys, 'frozen') and hasattr(sys, '_MEIPASS'):
    print('running in a PyInstaller bundle')
else:
    print('running in a normal Python process')
```

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. For a one-file bundle, this is the path to the temporary folder created by the bootloader (see *How the One-File Program Works*).

When your app is running, it may need to access data files in one of the following 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 variables for these uses.

#### 1.7.1 Using *__file__*

When your program is not bundled, the Python variable *__file__* refers to the current path of the module it is contained in. When importing a module from a bundled script, the *PyInstaller* bootloader will set the module’s *__file__* attribute to the correct path relative to the bundle folder.

For example, if you import `mypackage.mymodule` from a bundled script, then the *__file__* attribute of that module will be `sys._MEIPASS + 'mypackage/mymodule.pyc'`. So if you have a data file at `mypackage/file.dat` that you added to the bundle at `mypackage/file.dat`, the following code will get its path (in both the non-bundled and the bundled case):
```python
from os import path
def path_to_dat = path.join(path.dirname(__file__), 'file.dat')
```

In the bundled main script itself the above might not work, as it is unclear where it resides in the package hierarchy. So in when trying to find data files relative to the main script, `sys._MEIPASS` can be used. The following will get the path to a file `other-file.dat` next to the main script if not bundled and in the bundle folder if it is bundled:

```python
from os import path
import sys
bundle_dir = getattr(sys, '_MEIPASS', path.abspath(path.dirname(__file__)))
path_to_dat = path.join(bundle_dir, 'other-file.dat')
```

### 1.7.2 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:

```python
#!/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())
```

### 1.7.3 LD_LIBRARY_PATH / LIBPATH considerations

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

Chapter 1. What’s New This Release
If it exists, 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.

```python
env = dict(os.environ)  # make a copy of the environment
lp_key = 'LD_LIBRARY_PATH'  # for GNU/Linux and *BSD.
lp_orig = env.get(lp_key + '_ORIG')
if lp_orig is not None:
    env[lp_key] = lp_orig  # restore the original, unmodified value
else:
    # This happens when LD_LIBRARY_PATH was not set.
    # Remove the env var as a last resort:
    env.pop(lp_key, None)
p = Popen(system_cmd, ..., env=env)  # create the process
```

### 1.8 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`

### 1.8.1 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:

```python
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.

1.8.2 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:

```python
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:

```python
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:
You can also include the entire contents of a folder:

```python
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 `pkgutil.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:

```python
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()`:

```python
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:

```python
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:

```python
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.

**1.8.3 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:

```python
options = [ ('v', None, 'OPTION'), ('W ignore', None, 'OPTION') ]
a = Analysis( ... )
```

(continues on next page)
1.8.4 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:

```python
app = BUNDLE(exe,
             name='myscript.app',
             icon=None,
             bundle_identifier=None,
             info_plist={
                 'NSPrincipalClass': 'NSApplication',
                 'NSAppleScriptEnabled': False,
                 'CFBundleDocumentTypes': [
                     {
                         'CFBundleTypeName': 'My File Format',
                         'CFBundleTypeIconFile': 'MyFileIcon.icns',
                         'LSItemContentTypes': ['com.example.myformat'],
                         'LSHandlerRank': 'Owner'
                     }
                 ]
             })
```

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. Its argument should be a Python dict with keys and values to be included in the `Info.plist` file. *PyInstaller* creates `Info.plist` from the `info_plist` dict using the Python Standard Library module `plistlib`. `plistlib` can handle nested Python objects (which are translated to nested XML), and translates Python data types to the proper `Info.plist` XML types. Here’s an example:

```python
app = BUNDLE(exe,
             name='myscript.app',
             icon=None,
             bundle_identifier=None,
             info_plist={
                 'NSPrincipalClass': 'NSApplication',
                 'NSAppleScriptEnabled': False,
                 'CFBundleDocumentTypes': [
                     {
                         'CFBundleTypeName': 'My File Format',
                         'CFBundleTypeIconFile': 'MyFileIcon.icns',
                         'LSItemContentTypes': ['com.example.myformat'],
                         'LSHandlerRank': 'Owner'
                     }
                 ],
             })
```

In the above example, the key/value `NSPrincipalClass': 'NSApplication'` is necessary to allow Mac OS X to render applications using retina resolution. The key `NSAppleScriptEnabled` is assigned the Python boolean `False`, which will be output to `Info.plist` properly as `<false/>`. Finally the key `CFBundleDocumentTypes` tells Mac OS X what filetypes your application supports (see Apple document types).
1.8.5 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 an 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.

### 1.8.6 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.
**SPEC** 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/warn-myscript.txt`.

### 1.9 Notes about specific Features

This sections describes details about specific features. For a full list of features please refer to the website.

#### 1.9.1 Ctypes Dependencies

Ctypes is a foreign function library for Python, that allows calling functions present in shared libraries. Those libraries are not imported as Python packages, because they are not picked up via Python imports: their path is passed to ctypes instead, which deals with the shared library directly; this caused <1.4 PyInstaller import detect machinery to miss those libraries, failing the goal to build self-contained PyInstaller executables:

```python
from ctypes import *
# This will pass undetected under PyInstaller detect machinery,
# because it' s not a direct import.
handle = CDLL("/usr/lib/library.so")
handle.function_call()
```

**Solution in PyInstaller**

PyInstaller contains a pragmatic implementation of Ctypes dependencies: it will search for simple standard usages of ctypes and automatically track and bundle the referenced libraries. The following usages will be correctly detected:

```python
CDLL("library.so")
WinDLL("library.so")
cypes.DLL("library.so")
cdll.library # Only valid under Windows - a limitation of ctypes, not PyInstaller's
windll.library # Only valid under Windows - a limitation of ctypes, not PyInstaller's
cdll.LoadLibrary("library.so")
windll.LoadLibrary("library.so")
```

More in detail, the following restrictions apply:

- **only libraries referenced by bare filenames (e.g. no leading paths) will be handled**: handling absolute paths would be impossible without modifying the bytecode as well (remember that while running frozen, ctypes would keep searching the library at that very absolute location, whose presence on the host system nobody can guarantee), and relative paths handling would require recreating in the frozen executable the same hierarchy of directories leading to the library, in addition of keeping track of which the current working directory is;

- **only library paths represented by a literal string will be detected and included in the final executable**: PyInstaller import detection works by inspecting raw Python bytecode, and since you can pass the library path to ctypes using a string (that can be represented by a literal in the code, but also by a variable, by the return value of an arbitrarily complex function, etc...), it’s not reasonably possible to detect all ctypes dependencies;

- **only libraries referenced in the same context of ctypes’ invocation will be handled**.
We feel that it should be enough to cover most ctypes' usages, with little or no modification required in your code.

If PyInstaller does not detect a library, you can add it to your bundle by passing the respective information to --add-binary option or listing it in the .spec-file. If your frozen application will be able to pick up the library at run-time can not be guaranteed as it depends on the detailed implementation.

**Gotchas**

The ctypes detection system at *Analysis time* is based on ctypes.util.find_library(). This means that you have to make sure that while performing Analysis and running frozen, all the environment values find_library() uses to search libraries are aligned to those when running un-frozen. Examples include using LD_LIBRARY_PATH or DYLD_LIBRARY_PATH to widen find_library() scope.

### 1.9.2 SWIG support

PyInstaller tries to detect binary modules created by SWIG. This detection requires:

- The Python wrapper module must be imported somewhere in your application (or by any of the modules it uses).
- The wrapper module must be available as source-code and it's first line must contain the text automatically generated by SWIG.
- The C-module must have the same name as the wrapper module prefixed with an underscore (_). (This is a SWIG restriction already.)
- The C-module must sit just beside the wrapper module (thus a relative import would work).

Also some restrictions apply for Python 3, due to the way the SWIG wrapper is implemented:

- The C-module will become a global module. As a consequence, you can not use two SWIG modules with the same basename (e.g. pkg1._cmod and pkg2._cmod), as one would overwrite the other.

### 1.9.3 Cython support

PyInstaller can follow import statements that refer to Cython C object modules and bundle them – like for any other module implemented in C.

But – again, as for any other module implemented in C – PyInstaller can not determine if the Cython C object module is importing some Python module. These will typically show up as in a traceback like this (mind the .pyx extension):

```
Traceback (most recent call last):
[...]
File "myapp\cython_module.pyx", line 3, in init myapp.cython_module
ModuleNotFoundError: No module named 'csv'
```

So if you are using a Cython C object module, which imports Python modules, you will have to list these as --hidden-import.

### 1.10 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 itself, or your bundled app, terminates with a Python traceback. Then please consider the following actions in sequence, before asking for technical help.
1.10.1 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!

1.10.2 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/warn-name.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/warn-name.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

On each run PyInstaller writes a cross-referencing file about dependencies into the build folder: 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 ones. 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.

If you specify --log-level=DEBUG to the pyinstaller command, PyInstaller additionally generates a GraphViz input file representing the dependency graph. The file is build/name/graph-name.dot in the work-path= directory. You can process it with any GraphViz command, e.g. 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 GNU/Linux or DYLD_LIBRARY_PATH in OS X.

Getting Debug Messages

The --debug=all option (and its choices) provides a significant amount of diagnostic information. 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 build the app with the --debug=imports option (see Getting Debug Messages above), which will pass the -v (verbose imports) flag to the embedded Python interpreter. 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 for your production version.

Figuring Out Why Your GUI Application Won’t Start

If you are using the --windowed option, your bundled application ay fail to start with an error message like Failed to execute script my_gui. In this case, you will want to get more verbose output to find out what is going on.

• For Mac OS, you can run your application on the command line, i.e.‘‘./dist/my_gui‘‘ in Terminal instead of clicking on my_gui.app.

• For Windows, you will need to re-bundle your application without the --windowed option. Then you can run the resulting executable from the command line, i.e.: my_gui.exe.

• For Unix and GNU/Linux there is no --windowed option. Anyway, if a your GUI application fails, you can run your application on the command line, i.e. ./dist/my_gui.

This should give you the relevant error that is preventing your application from initializing, and you can then move on to other debugging steps.

Operation not permitted error

If you use the --onefile and it fails to run you program with error like:

```
./hello: error while loading shared libraries: libz.so.1: failed to map segment from shared object: Operation not permitted
```
This can be caused by wrong permissions for the /tmp directory (e.g. the filesystem is mounted with noexec flags).

A simple way to solve this issue is to set, in the environment variable TMPDIR, a path to a directory in a filesystem mounted without noexec flags, e.g.:

```
export TMPDIR=/var/tmp/
```

### 1.10.3 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__, imp.find_module() 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 --debug=imports flag (see 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 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:

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:

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

1.10.4 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 https://github.com/pyinstaller/pyinstaller/archive/develop.zip
```

1.10.5 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.

1.11 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
1.11.1 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:

A. First process: bootloader starts.
   1. If one-file mode, extract bundled files to tempath/_MEIxXXXXX.
   2. Modify various environment variables:
      - GNU/Linux: If set, save the 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.
   3. Set up to handle signals for both processes.
   4. Run the child process.
   5. Wait for the child process to finish.
   6. If one-file mode, delete tempath/_MEIxXXXXX.

B. Second process: bootloader itself started as a child process.
   1. On Windows set the activation context.
   2. Load the Python dynamic library. The name of the dynamic library is embedded in the executable file.
   4. 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`.

1.11.2 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:

<table>
<thead>
<tr>
<th><code>typecode</code></th>
<th><code>description</code></th>
<th><code>name</code></th>
<th><code>path</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>'DATA'</td>
<td>Arbitrary files.</td>
<td>Run-time name.</td>
<td>Full path name in build.</td>
</tr>
<tr>
<td>'BINARY'</td>
<td>A shared library.</td>
<td>Run-time name.</td>
<td>Full path name in build.</td>
</tr>
<tr>
<td>'EXTENSION'</td>
<td>A binary extension to Python.</td>
<td>Run-time name.</td>
<td>Full path name in build.</td>
</tr>
<tr>
<td>'OPTION'</td>
<td>A Python run-time option.</td>
<td>Option code</td>
<td>ignored.</td>
</tr>
</tbody>
</table>

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:

```python
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:

```python
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:

```python
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).

1.11.3 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.

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, GNU/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.

Using pyi-archive_viewer

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

    pyi-archive_viewer archivefile
Fig. 1: Structure of the ZlibArchive
Fig. 2: Structure of the CArchive
Fig. 3: Structure of the Self Extracting Executable
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 .PYZ-00.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.
- **-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.

### 1.11.4 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.

### 1.11.5 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 bytecode 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 GNU/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
```
1.12 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:

```python
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:

```python
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.

1.12.1 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.

### 1.12.2 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:

```python
hiddenimports = ['_gdbm', 'socket', 'h5py.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:

```python
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:

```python
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:

```python
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:

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

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

```python
binaries = collect_dynamic_libs('zmq')
```
1.12.3 Useful Items in PyInstaller.compat

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

```python
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_py35, is_py36, is_py37**: True when the current version of Python is at least 3.5, 3.6, or 3.7 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 GNU/Linux system (`sys.platform.startswith('linux')`).
- **is_solar**: 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:

```python
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 file: `hook-cryptography.py` for an example.

1.12.4 Useful Items in PyInstaller.utils.hooks

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

```python
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:

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

mpl_data_dir = exec_statement(
    "import matplotlib; print(matplotlib.get_data_path())"
)
```

(continues on next page)
"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:

```python
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:

```python
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 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_all**, 'package-name', include_py_files=False):

Given a package name as a string, this function returns a tuple of datas, binaries, hiddenimports containing all data files, binaries, and modules in the given package, including any modules specified in the requirements for the distribution of this module. The value of include_py_files is passed directly to collect_data_files.

Typical use: datas, binaries, hiddenimports = collect_all('my_module_name'). For example, hook-gevent.py invokes collect_all, which gathers:

- All data files, such as __greenlet_primitives.pxd, __hub_local.pxd, and many, many more.
- All binaries, such as __greenlet_primitives.cp37-win_amd64.pyd (on a Windows 64-bit install) and many, many more.
- All modules in `gevent`, such as `gevent.threadpool`, `gevent._semaphore`, and many, many more.
- All requirements. `pip show gevent` gives Requires: `cffi`, `greenlet`. Therefore, the `cffi` and `greenlet` modules are included.

```python
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 a 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 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`.

```python
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`.
```

```python
collect_data_files( 'module-name', include_py_files=False, subdir=None ): 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.)

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

```python
get_module_file_attribute( 'module-name' ): Return the absolute path to `module-name`, a fully-qualified module name. Example:
```
```python
nacl_dir = os.path.dirname(get_module_file_attribute('nacl'))
```

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

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

```python
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.

1.12.5 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.

1.12.6 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.

### 1.12.7 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`:

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

### 1.12. Understanding *PyInstaller* Hooks

51
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.

1.13 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 for the current platform (operating-system and word-size), 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,

- Download and install Python, which is required for running :command:waf,
- git clone or download the source (see the Download section on the web-site),
- cd into the folder where you cloned or unpacked the source to,
- cd bootloader, and
- make the bootloader with: python ./waf all,
- test the build by ref:running (parts of) the test-suite <running-the-test-suite>.

This will produce the bootloader executables for your current platform (of course, for Windows these files will have the .exe extension):

- ../PyInstaller/bootloader/OS_ARCH/run,
- ../PyInstaller/bootloader/OS_ARCH/run_d,
- ../PyInstaller/bootloader/OS_ARCH/runw (OS X and Windows only), and
- ../PyInstaller/bootloader/OS_ARCH/runw_d (OS X and Windows only).

The bootloaders architecture defaults to the machine’s one, but can be changed using the --target-arch= option – given the appropriate compiler and development files are installed. E.g. to build a 32-bit bootloader on a 64-bit machine, run:

```sh
python ./waf all --target-arch=32bit
```

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

Supported platforms are

- GNU/Linux (using gcc)
- Windows (using Visual C++ or MinGW’s gcc)
- Mac OS X (using clang)

Contributed platforms are

- AIX (using gcc or xlc)
- HP-UX (using gcc or xlc)
1.13 Building the Bootloader

1.13.1 Building for GNU/Linux

Development Tools

For building the bootloader you’ll need a development environment. You can run the following to install everything required:

- On Debian- or Ubuntu-like systems:

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

- On Fedora, RedHat and derivates:

  ```
  sudo yum groupinstall "Development Tools"
  ```

- For other Distributions please consult the distributions documentation.

Now you can build the bootloader as shown above.

Alternatively you may want to use the `linux64` build-guest provided by the Vagrantfile (see below).

Building Linux Standard Base (LSB) compliant binaries (optional)

By default, the bootloaders on GNU/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 binaries, you can do so by specifying `--lsb` on the waf command line, as follows:

```
python ./waf distclean all --lsb
```

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

1.13.2 Building for Mac OS X

On Mac OS X please install Xcode, Apple’s suite of tools for developing software for Mac OS X. This will get you the `clang` compiler. Any version suitable for your platform should be fine. Xcode can be also installed from your Mac OS X Install DVD.

Now you can build the bootloader as shown above.

Alternatively you may want to use the `darwin64` build-guest provided by the Vagrantfile (see below).

---

1 Linux Standard Base (LSB) is a set of open standards that should increase compatibility among GNU/Linux distributions. Unfortunately it is not widely adopted and both Debian and Ubuntu dropped support for LSB in autumn 2015. Thus PyInstaller bootloader are no longer provided as LSB binary.
Cross-Building for Mac OS X

For cross-compiling for OS X you need the Clang/LLVM compiler, the *cctools* (ld, lipo, ...), and the OSX SDK. Clang/LLVM is a cross compiler by default and is available on nearly every GNU/Linux distribution, so you just need a proper port of the cctools and the OS X SDK.

This is easy to get and needs to be done only once and the result can be transferred to your build-system. The build-system can then be a normal (somewhat current) GNU/Linux system.²

Preparation: Get SDK and Build-tools

For preparing the SDK and building the cctools, we use the very helpful scripts from the *OS X Cross* <https://github.com/tpoechtrager/osxcross> toolchain. If you're interested in the details, and what other features OS X Cross offers, please refer to its homepage.

Side-note: For actually accessing the OS X disk image file (.dmg), *darling-dmg* is used. It allows mounting .dmg's under GNU/Linux via FUSE.

For saving you reading OSXCross' documentation we prepared a virtual box description performing all required steps. If you are interested in the precise commands, please refer to *packages_osxcross_debianoid*, *prepare_osxcross_debianoid*, and *build_osxcross* in the Vagrantfile.

Please proceed as follows:

1. Download XCode 7.3.x <https://developer.apple.com/downloads/index.action?name=Xcode%207.3> and save it to bootloader/sdks/osx/. You will need to register an Apple ID, for which you may use a disposable e-mail-address, to search and download the files.

   Please make sure that you are complying to the license of the respective package.

2. Use the Vagrantfile to automatically build the SDK and tools:

   ```
   vagrant up build-osxcross && vagrant halt build-osxcross
   ```

   This should create the file *bootloader/sdks/osx/osxcross.tar.xz*, which will then be installed on the build-system.

   If for some reason this fails, try running *vagrant provision build-osxcross*.

3. This virtual machine is no longer used, you may now want to discard it using *vagrant destroy build-osxcross*.

Building the Bootloader

Again, simply use the Vagrantfile to automatically build the OS X bootloaders:

```
export TARGET=OSX   # make the Vagrantfile build for OS X
vagrant up linux64 && vagrant halt linux
```

This should create the bootloaders in */../PyInstaller/bootloader/Darwin-*/.

   If for some reason this fails, try running *vagrant provision linux64*.

   3. This virtual machine is no longer used, you may now want to discard it using:

² Please keep in mind that to avoid problems, the system you are using for the preparation steps should have the same architecture (and possible the same GNU/Linux distribution version) as the build-system.
4. If you are finished with the OS X bootloaders, unset `TARGET` again:

```bash
unset TARGET
```

If you don’t want to use the build-guest provided by the Vagrant file, perform the following steps (see `build_bootloader_target_osx` in the Vagrantfile):

```bash
mkdir -p ~/osxcross
tar -C ~/osxcross --xz -xf /vagrant/sdks/osx/osxcross.tar.xz
PATH=~/osxcross/bin/:$PATH
python ./waf all CC=x86_64-apple-darwin15-clang
python ./waf all CC=i386-apple-darwin15-clang
```

## 1.13.3 Building for Windows

The pre-compiled bootloader coming with PyInstaller are self-contained static executable that imposes no restrictions on the version of Python being used.

When building the bootloader yourself, you have to carefully choose between three options:

1. Using the Visual Studio C++ compiler.
   - This allows creating self-contained static executables, which can be used for all versions of Python. This is why the bootloaders delivered with PyInstaller are built using Visual Studio C++ compiler.
   - You can use any Visual Studio version that is convenient (as long as it’s supported by the waf build-tool).

2. Using the MinGW-w64 suite.
   - This allows to create smaller, dynamically linked executables, but requires to use the same level of Visual Studio as was used to compile Python. So this bootloader will be tied to a specific version of Python.
   - The reason for this is, that unlike Unix-like systems, Windows doesn’t supply a system standard C library, leaving this to the compiler. But Mingw-w64 doesn’t have a standard C library. Instead it links against msvcrt.dll, which happens to exist on many Windows installations – but it’s not guaranteed to exist.

3. Using cygwin and MinGW.
   - This will create executables for cygwin, not for ‘plain’ Windows.

In all cases you may want

- to set the path to include python, e.g. `set PATH=%PATH%;c:\python35`,
- to peek into the Vagrantfile or `../appveyor.yml` to learn how we are building.

You can also build the bootloaders for cygwin.

### Build using Visual Studio C++

- With our `wscript` file, you don’t need to run `vcvarsall.bat` to `switch` the environment between VC++ installations and target architecture. The actual version of C++ does not matter and the target architecture is selected by using the `--target-arch=` option.

---

3 This description seems to be technically incorrect. I ought to depend on the C++ run-time library. If you know details, please open an issue.
• If you are not using Visual Studio for other work, installing only the standalone C++ build-tools might be the best option as it avoids bloating your system with stuff you don’t need (and saves a lot if installation time).

**Hint:** We recommend installing the build-tools software using the chocolatey package manager. While at a first glance it looks like overdose, this is the easiest way to install the C++ build-tools. It comes down to two lines in an administrative powershell:

```bash
... one-line-install as written on the chocolatey homepage
choco install -y python vcbuildtools
```

• Useful Links:
  
  – Microsoft Visual C++ Build-Tools 2015
  

After installing the C++ build-tool you can build the bootloader as shown above.

### Build using MinGW-w64

Please be aware of the restrictions mentioned above.

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

Note: Please mind that using cygwin’s python or MinGW when running ./waf will create executables for cygwin, not for Windows.

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

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

Now you can build the bootloader as shown above. If you have installed both Visual C++ and MinGW, you might need to add run python ./waf --gcc all.

### Build using cygwin and MinGW

Please be aware that this will create executables for cygwin, not for ‘plain’ Windows.

Use cygwin’s `setup.exe` to install python and mingw.

Now you can build the bootloader as shown above.

### 1.13.4 Vagrantfile Virtual Machines

PyInstaller maintains a set of virtual machine description for testing and (cross-) building. For managing these boxes, we use `vagrant`.

All guests\(^4\) will automatically build the bootloader when running `vagrant up GUEST` or `vagrant provision GUEST`. They will build both 32- and 64-bit bootloaders.

\(^4\) Except of guest `osxcross`, which will build the OS X SDK and cctools as described in section *Cross-Building for Mac OS X*.
All guests (except of darwin64), when building the bootloaders, are sharing the PyInstaller distribution folder and will put the built executables onto the build-host (into ../PyInstaller/bootloader/).

Most boxes requires two Vagrant plugins to be installed:

```
vagrant plugin install vagrant-reload vagrant-scp
```

Example usage:

```
vagrant up linux64    # will also build the bootloader
vagrant halt linux64  # or `destroy`
# verify the bootloader has been rebuild
git status ../PyInstaller/bootloader/
```

You can pass some parameters for configuring the Vagrantfile by setting environment variables, like this:

```
GUI=1 TARGET=OSX vagrant up darwin64
```

or like this:

```
export TARGET=OSX
vagrant provision linux64
```

We currently provide this guests:

- **linux64** GNU/Linux (some recent version) used to build the GNU/Linux bootloaders.
  
  - If TARGET=OS is set, cross-builds the bootloaders for OS X (see Cross-Building for Mac OS X).
  
  - If TARGET=WINDOWS is set, cross-builds the bootloaders for Windows using mingw. Please have in mind that this imposes the restrictions mentioned above.
  
  - Otherwise (which is the default) bootloaders for GNU/Linux are build.

- **darwin64** Mac OS X ‘Yosemite’ – not actually used by the PyInstaller team, but provided for testing.
  
  This guest, when building the bootloaders, does not put the built executables onto the build-host. You need to fetch them using:

```
vagrant plugin install vagrant-scp vagrant-reload # required only once
vagrant scp -a darwin64:/vagrant/PyInstaller/bootloader/Darwin-* ../PyInstaller/bootloader/
```

This is due the fact that this machine doesn’t include the Virtualbox guest additions and thus doesn’t support shared folders.

- **windows10** Windows 10, used for building the Windows bootloaders using Visual C++.
  
  - If MINGW=1 is set, the bootloaders will be build using MinGW. Please be aware of the restrictions mentioned above.

  **Note:** The Windows box uses password authentication, so in some cases you need to enter the password (which is Passw0rd!).

- **build-osxcross** GNU/Linux guest used to build the OS X SDK and cctools as described in section Cross-Building for Mac OS X.
1.14 Changelog for PyInstaller

1.14.1 3.5 (2019-07-09)

Features

- (Windows) Force --windowed option if first script is a .pyw file. This might still be overwritten in the spec-file. (#4001)
- Add support for relative paths for icon-files, resource-files and version-resource-files. (#3333, #3444)
- Add support for the RedHat Software Collections (SCL) Python 3.x. (#3536, #3881)
- Install platform-specific dependencies only on that platform. (#4166, #4173)
- New command-line option --upx-exclude, which allows the user to prevent binaries from being compressed with UPX. (#3821)

Bugfix

- (conda) Fix detection of conda/anaconda platform.
- (GNU/Linux) Fix Anaconda Python library search. (#3885, #4015)
- (Windows) Fix UAC in one-file mode by embedding the manifest. (#1729, #3746)
- (Windows\Py3.7) Now able to locate pylib when VERSION.dll is listed in python.exe PE Header rather than pythonXY.dll (#3942, #3956)
- Avoid errors if PyQt5 or PySide2 is referenced by the modulegraph but isn’t importable. (#3997)
- Correctly parse the --debug=import, --debug=bootloader, and --debug=noarchive command-line options. (#3808)
- Don’t treat PyQt5 and PySide2 files as resources in an OS X windowed build. Doing so causes the resulting frozen app to fail under Qt 5.12. (#4237)
- Explicitly specify an encoding of UTF-8 when opening all text files. (#3605)
- Fix appending the content of datas in a spec files to binaries instead of the internal datas. (#2326, #3694)
- Fix crash when changing from --onefile to --onedir on consecutive runs. (#3662)
- Fix discovery of Qt paths on Anaconda. (#3740)
- Fix encoding error raised when reading a XML manifest file which includes non-ASCII characters. This error inhibited building an executable which has non-ASCII characters in the filename. (#3478)
- Fix inputs to QCoreApplication constructor in Qt5LibraryInfo. Now the core application’s initialization and finalization in addition to system-wide and application-wide settings is safer. (#4121)
- Fix installation with pip 19.0. (#4003)
- Fixes PE-file corruption during version update. (#3142, #3572)
- In the fake ‘site’ module set USER_BASE to empty string instead of None as Jupyter Notebook requires it to be a 'str'. (#3945)
- Query PyQt5 to determine if SSL is supported, only adding SSL DLLs if so. In addition, search the path for SSL DLLs, instead of looking in Qt’s BinariesPath. (#4048)
- Require pywin32-ctypes version 0.2.0, the minimum version which supports Python 3.7. (#3763)
• Use pkgutil instead of filesystem operations for interacting with the modules. (#4181)

Incompatible Changes

• PyInstaller is no longer tested against Python 3.4, which is end-of-life.
• Functions compat.architecture(), compat.system() and compat.machine() have been replace by variables of the same name. This avoids evaluating the save several times.
• Require an option for the --debug argument, rather than assuming a default of all. (#3737)

Hooks

• Added hooks for aliyunsdkcore (#4228), astropy (#4274), BTrees (#4239), dateparser.utils.strptime (#3790), faker (#3989, #4133), gooeys (&3773), GtkSourceView (#3893), imageio_ffmpeg (#4051), importlib_metadata and importlib_resources (#4095), jspnpath_rw_ext (#3841), jupyterlab (#3951), lz4 (#3710), magic (#4267), nante (#3860), nbconvert (#3947), nbtime (#3949), nbformat (#3946), notebook (#3950), pendulum (#3906), pysoundfile (#3844), python-docx (#2574, #3848), python-wavefile (#3785), pytzdata (#3906), PyWavelets pywt (#4120), pywebview (#3771), radicale (#4109), rdffib (#3708), resampy (#3702), sqlalchemy-migrate (#4250), textdistance (#4239), tcod (#3622), ttkthemes (#4105), and umap-learn (#4165).
• Add runtime hook for certifi. (#3952)
• Updated hook for ‘notebook’ to look in all Jupyter paths reported by jupyter_core. (#4270)
• Fixed hook for ‘notebook’ to only include directories that actually exist. (#4270)
• Fixed pre-safe-import-module hook for setuptools.extern.six. (#3806)
• Fixed QtWebEngine hook on OS X. (#3661)
• Fixed the QtWebEngine hook on distributions which don’t have a NSS subdir (such as Archlinux) (#3758)
• Include dynamically-imported backends in the eth_hash package. (#3681)
• Install platform-specific dependencies only on that platform. (#4168)
• Skip packaging PyQt5 QML files if the QML directory doesn’t exist. (#3864)
• Support ECC in PyCryptodome. (#4212, #4229)
• Updated PySide2 hooks to follow PyQt5 approach. (#3655, #3689, #3724, #4040, #4103, #4136, #4175, #4177, #4198, #4206)
• Updated the jsonschema hook for v3.0+. (#4100)
• Updated the Sphinx hook to correctly package Sphinx 1.8.

Bootloader

• Update bundled zlib library to 1.2.11 address vulnerabilities. (#3742)

Documentation

• Update the text produced by --help to state that the --debug argument requires an option. Correctly format this argument in the Sphinx build process. (#3737)
Project & Process

- Remove the PEP-518 “build-system” table from pyproject.toml to fix installation with pip 19.0.

PyInstaller Core

- Add support for folders in COLLECT and BUNDLE. (#3653)
- Completely remove pywin32 dependency, which has erratic releases and the version on pypi may no longer have future releases. Require pywin32-ctypes instead which is pure python. (#3728, #3729)
- modulegraph: Align with upstream version 0.17.
- Now prints a more descriptive error when running a tool fails (instead of dumping a trace-back). (#3772)
- Suppress warnings about missing UCRT dependencies on Win 10. (#1566, #3736)

Test-suite and Continuous Integration

- Fix Appveyor failures of test_stderr_encoding() and test_stdout_encoding() on Windows Python 3.7 x64. (#4144)
- November update of packages used in testing. Prevent pyup from touching test/requirements-tools. txt. (#3845)
- Rewrite code to avoid a RemovedInPytest4Warning: Applying marks directly to parameters is deprecated, please use pytest.param(..., marks=...) instead. (#3772)
- Run Travis tests under Xenial; remove the deprecated sudo: false tag. (#4140)
- Update the Markdown test to comply with Markdown 3.0 changes by using correct syntax for extensions.

1.14.2 3.4 (2018-09-09)

Features

- Add support for Python 3.7 (#2760, #3007, #3076, #3399, #3656), implemented by Hartmut Goebel.
- Improved support for Qt5-based applications (#3439). By emulating much of the Qt deployment tools’ behavior most PyQt5 variants are supported. However, Anaconda’s PyQt5 packages are not supported because its QLibraryInfo implementation reports incorrect values. CI tests currently run on PyQt5 5.11.2. Many thanks to Bryan A. Jones for taking this struggle.
- --debug now allows more debugging to be activated more easily. This includes bootloader messages, Python’s “verbose imports” and store collected Python files in the output directory instead of freezing. See pyinstaller --help for details. (#3546, #3585, #3587)
- Hint users to install development package for missing pyconfig.h. (#3348)
- In setup.py specify Python versions this distribution is compatible with.
- Make base_library.zip reproducible: Set time-stamp of files. (#2952, #2990)
- New command-line option --bootloader-ignore-signals to make the bootloader forward all signals to the bundle application. (#208, #3515)
- (OS X) Python standard library module plistlib is now used for generating the Info.plist file. This allows passing complex and nested data in info.plist. (#3532, #3541)
Bugfix

• Add missing `warnings` module to `base_library.zip`. (#3397, #3400)
• Fix and simplify search for libpython on Windows, msys2, cygwin. (#3167, #3168)
• Fix incompatibility with `pycryptodome` (a replacement for the apparently abandoned `pycrypto` library) when using encrypted `PYZ-archives`. (#3537)
• Fix race condition caused by the bootloader parent process terminating before the child is finished. This might happen e.g. when the child process itself plays with `switch_root`. (#2966)
• Fix wrong security alert if a filename contains `..`. (#2641, #3491)
• Only update resources of cached files when necessary to keep signature valid. (#2526)
• (OS X) Fix: App icon appears in the dock, even if `LSUIElement=True`. (#1917, #2075, #3566)
• (Windows) Fix crash when trying to add resources to Windows executable using the `--resource` option. (#2675, #3423)
• (Windows) Only update resources when necessary to keep signature valid (#3323)
• (Windows) Use UTF-8 when reading XML manifest file. (#3476)
• (Windows) `utils/win32`: trap invalid `--icon` arguments and terminate with a message. (#3126)

Incompatible Changes

• Drop support for Python 3.3 (#3288), Thanks to Hugo and xoviat.
• `--debug` now expects an (optional) argument. Thus using `... --debug script.py` will break. Use `... script.py --debug or ... --debug=all script.py` instead. Also `--debug=all` (which is the default if no argument is given) includes `noarchive`, which will store all collected Python files in the output directory instead of freezing them. Use `--debug=bootloader` to get the former behavior. (#3546, #3585, #3587)
• (minor) Change naming of intermediate build files and the `warn` file. This only effects 3rd-party tools (if any exists) relying on the names of these files.
• (minor) The destination path for `--add-data` and `--add-binary` must no longer be empty, use . instead. (#3066)
• (minor) Use standard path, not dotted path, for C extensions (Python 3 only).

Hooks

• New hooks for bokeh visualization library (#3607), Champlain, Clutter (#3443) dynaconf (#3641), flex (#3401), FMPy (#3589), gi.repository.xlib (#2634, #3396) google-cloud-translate, google-api-core (#3658), jedi (#3535, #3612), nltk (#3705), pandas (#2978, #2998, #2999, #3015, #3063, #3079), phonenumbers (#3381, #3558), pinyin (#2822), PySide.phonon, PySide.QtSql (#2859), pytorch (#3657), scipy (#2987, #3048), uvloop (#2898), web3, eth_account, eth_keyfile (#3365, #3373).
• Updated hooks for Cryptodome 3.4.8, Django 2.1, gevent 1.3. Crypto (support for PyCryptodome) (#3424), Gst and GdkPixbuf (to work on msys2, #3257, #3387), sphinx 1.7.1, setuptools 39.0.
• Updated hooks for PyQt5 (#1930, #1988, #2141, #2156, #2220, #2518, #2566, #2573, #2577, #2857, #2924, #2976, #3175, #3211, #3233, #3308, #3338, #3417, #3439, #3458, #3505), among others:
  – All QML is now loaded by `QtCore.QMetaEngine`. 

1.14. Changelog for PyInstaller
– Improve error reporting when determining the PyQt5 library location.
– Improved method for finding `qt.conf`.
– Include OpenGL fallback DLLs for PyQt5. (#3568).
– Place PyQt5 DLLs in the correct location (#3583).
• Fix hooks for cryptodome (#3405), PySide2 (style mismatch) (#3374, #3578)
• Fix missing SSL libraries on Windows with `PyQt5.QtNetwork`. (#3511, #3520)
• Fix `zmq` on Windows Python 2.7. (#2147)
• (GNU/Linux) Fix hook `usb`: Resolve library name reported by `usb.backend`. (#2633, #2831, #3269)
• Clean up the USB hook logic.

Bootloader

• Forward all signals to the child process if option `pyi-bootloader-ignore-signals` to be set in the archive. (#208, #3515)
• Use `waitpid` instead of `wait` to avoid the bootloader parent process gets signaled. (#2966)
• (OS X) Don’t make the application a GUI app by default, even in `--windowed` mode. Not enforcing this programmatically in the bootloader allows to control behavior using `Info.plist` options - which can be set in PyInstaller itself or in the `.spec`-file. (#1917, #2075, #3566)
• (Windows) Show respectively print utf-8 debug messages ungarbled. (#3477)
• Fix `setenv()` call when `HAVE_UNSETENV` is not defined. (#3722, #3723)

Module Loader

• Improved error message in case importing an extension module fails. (#3017)

Documentation

• Fix typos, smaller errors and formatting errors in documentation. (#3442, #3521, #3561, #3638)
• Make clear that `--windowed` is independent of `--onedir`. (#3383)
• Mention imports using `import imp.find_module()` are not detected.
• Reflect actual behavior regarding `LD_LIBRARY_PATH`. (#3236)
• (OS X) Revise section on `info.plist` for `plistlib` functionality and use an example more aligned with real world usage. (#3532, #3540, #3541)
• (developers) Overhaul guidelines for commit and commit-messages. (#3466)
• (developers) Rework developer's quick-start guide.

Project & Process

• Add a `pip requirements.txt` file.
• Let `pyup` update package requirements for “Test – Libraries” every month only.
• Use `towncrier` to manage the change log entries. (#2756, #2837, #3698)
PyInstaller Core

- Add `requirements_for_package()` and `collect_all()` helper functions for hooks.
- Add an explanatory header to the warn-file, hopefully reducing the number of those posting the file to the issue tracker.
- Add module `enum` to `base_library.zip`, required for module `re` in Python 3.6 (and `re` is required by `warnings`).
- Always write the `warn` file.
- Apply `format_binaries_and_datas()` (which converts hook-style tuples into TOC-style tuples) to binaries and datas added through the hook api.
- Avoid printing a useless exceptions in the `get_module_file_attribute()` helper function.
- Don’t gather Python extensions in `collect_dynamic_libc()`.
- Fix several ResourceWarnings and DeprecationWarnings (#3677)
- Hint users to install necessary development packages if, in `format_binaries_and_datas()`, the file not found is `pyconfig.h`. (#1539, #3348)
- Hook helper function `is_module_satisfies()` returns `False` for packages not found. (#3428, #3481)
- Read data for cache digest in chunks. (#3281)
- Select correct file extension for C-extension file-names like `libzmq.cp36-win_amd64.pyd`.
- State type of import (conditional, delayed, etc.) in the `warn` file again.
- (modulegraph) Unbundle `altgraph` library, use from upstream. (#3058)
- (OS X) In --console mode set `LSBackgroundOnly=True` in ‘Info.plist’ to hide the app-icon in the dock. This can still be overruled by passing `info_plist` in the `.spec`-file. (#1917, #3566)
- (OS X) Use the python standard library `plistlib` for generating the `Info.plist` file. (#3532, #3541)
- (Windows) Completely remove `pywin32` dependency, which has erratic releases and the version on pypi may no longer have future releases. Require `pywin32-ctypes` instead, which is pure python. (#3141)
- (Windows) Encode manifest before updating resource. (#3423)
- (Windows) Make import compatible with python.net, which uses an incompatible signature for `__import__`. (#3574)

Test-suite and Continuous Integration

- Add script and dockerfile for running tests in docker. (Contributed, not maintained) (#3519)
- Avoid log messages to be written (and captured) twice.
- Fix decorator `skipif_no_compiler`.
- Fix the test for the “W” run-time Python option to verify module `warnings` can actually be imported. (#3402, #3406)
- Fix unicode errors when not capturing output by pytest.
- Run `pyinstaller --h` to verify it works.
- `test_setuptools_nspkg` no longer modifies source files.
- Appveyor:
– Add documentation for Appveyor variables used to appveyor.yml.
– Significantly clean-up appveyor.yml (#3107)
– Additional tests produce > 1 hour runs. Split each job into two jobs.
– Appveyor tests run on 2 cores; therefore, run 2 jobs in parallel.
– Reduce disk usage.
– Split Python 2.7 tests into two jobs to avoid the 1 hour limit.
– Update to use Windows Server 2016. (#3563)

• Travis
  – Use build-stages.
  – Clean-up travis.yml (#3108)
  – Fix Python installation on OS X. (#3361)
  – Start a X11 server for the “Test - Libraries” stage only.
  – Use target python interpreter to compile bootloader to check if the build tool can be used with that this Python version.

Bootloader build

• Print invoking python version when compiling.
• Update waf build-tool to 2.0.9 and fix our wscript for waf 2.0.
• (GNU/Linux) When building with --debug turn of FORTIFY_SOURCE to ease debugging.

Known Issues

• Anaconda’s PyQt5 packages are not supported because its QlibraryInfo implementation reports incorrect values.
• All scripts frozen into the package, as well as all run-time hooks, share the same global variables. This issue exists since v3.2 but was discovered only lately, see #3037. This may lead to leaking global variables from run-time hooks into the script and from one script to subsequent ones. It should have effects in rare cases only, though.
• Data-files from wheels, unzipped eggs or not ad egg at all are not included automatically. This can be worked around using a hook-file, but may not suffice when using --onefile and something like python-daemon.
• The multipackage (MERGE) feature (#1527) is currently broken.
• (OSX) Support for OpenDocument events (#1309) is broken.
• (Windows) With Python 2.7 the frozen application may not run if the user-name (more specifically %TEMPDIR%) includes some Unicode characters. This does not happen with all Unicode characters, but only some and seems to be a windows bug. As a work-around please upgrade to Python 3 (#2754, #2767).
• (Windows) For Python >= 3.5 targeting Windows < 10, the developer needs to take special care to include the Visual C++ run-time .dlls. Please see the section Platform-specific Notes in the manual. (#1566)
1.14.3 3.3.1 (2017-12-13)

Hooks

• Fix imports in hooks accessible_output and sound_lib (#2860).
• Fix ImportError for sysconfig for 3.5.4 Conda (#3105, #3106).
• Fix shapely hook for conda environments on Windows (#2838).
• Add hook for unidecode.

Bootloader

• (Windows) Pre-build bootloaders (and custom-build ones using MSVC) can be used on Windows XP again. Set minimum target OS to XP (#2974).

Bootloader build

• Fix build for FreeBSD (#2861, #2862).

PyInstaller Core

• Usage: Add help-message clarifying use of options when a spec-file is provided (#3039).
• Add printing infos on UnicodeDecodeError in exec_command(_all).
• (win32) Issue an error message on errors loading the icon file (#2039).
• (aarch64) Use correct bootloader for 64-bit ARM (#2873).
• (OS X) Fix replacement of run-time search path keywords (@... ) (#3100).

Modulegraph

– Fix recursion too deep errors cause by reimporting SWIG-like modules (#2911, #3040, #3061).
– Keep order of imported identifiers.

Test-suite and Continuous Integration

• In Continuous Integration tests: Enable flake8-diff linting. This will refuse all changed lines not following PEP 8.
• Enable parallel testing on Windows,
• Update requirements.
• Add more test cases for modulegraph.
• Fix a test-case for order of module import.
• Add test-cases to check scripts do not share the same global vars (see Known Issues).
Documentation

- Add clarification about treatment of options when a spec-file is provided (#3039).
- Add docs for running PyInstaller with Python optimizations (#2905).
- Add notes about limitations of Cython support.
- Add information how to handle undetected ctypes libraries.
- Add notes about requirements and restrictions of SWIG support.
- Add note to clarify what *binary files* are.
- Add a Development Guide.
- Extend “How to Contribute”.
- Add “Running the Test Suite”.
- Remove badges from the Readme (#2853).
- Update outdated sections in man-pages and other enhancements to the man-page.

Known Issues

- All scripts frozen into the package, as well as all run-time hooks, share the same global variables. This issue exists since v3.2 but was discovered only lately, see #3037. This may lead to leaking global variables from run-time hooks into the script and from one script to subsequent ones. It should have effects in rare cases only, though.
- Further see the *Known Issues for release 3.3*.

1.14.4 3.3 (2017-09-21)

- **Add Support for Python 3.6!** Many thanks to xiovat! (#2331, #2341)
- New command line options for adding data files (--datas, #1990) and binaries (--binaries, #703)
- Add command line option `--runtime-tmpdir`.
- Bootloaders for Windows are now build using MSVC and statically linked with the run-time-library (CRT). This solved a lot of issues related to .dlls being incompatible with the ones required by `python.dll`.
- Bootloaders for GNU/Linux are now officially no LSB binaries. This was already the case since release 3.1, but documented the other way round. Also the build defaults to non-LSB binaries now. (#2369)
- We improved and stabilized both building the bootloaders and the continuous integration tests. See below for details. Many thanks to all who worked on this.
- To ease solving issues with packages included wrongly, the html-file with a cross-reference is now always generated. It’s visual appearance has been modernized (#2765).

Incompatible changes

- Command-line option obsoleted several version ago are not longer handled gracefully but raise an error (#2413)
- Installation: PyInstaller removed some internal copies of 3rd-party packages. These are now taken from their official releases at PyPI (#2589). This results in PyInstaller to no longer can be used from just an unpacked archive, but needs to be installed like any Python package. This should effect only a few people, e.g. the developers.
- Following **PEP 527**, we only release one source archive now and decided to use `.tar.gz` (#2754).

## Hooks

- New and Updated hooks: accessible_output2 (#2266), ADIOS (#2096), CherryPy (#2112), PySide2 (#2471, #2744) (#2472), Sphinx (#2612, 2708) (#2708), appdir (#2478), clr (#2048), cryptodome (#2125), cryptography (#2013), dclab (#2657), django (#2037), django migrations (#1795), django.contrib (#2336), google.cloud, google.cloud.storage, gstreamer (#2603), imageio (#2696), langcodes (#2682), libaudioverse (#2709), mpl_toolkits (#2400), numba, llvmlite (#2113), openpyxl (#2113), openpyxl, pyopencl, pyproj (#2677), pytest (#2119), qtpy (#2617), redmine, requests (#2334), setuptools, setuptools (#2565), shapely (#2569), sound_lib (#2267), sysconfig, uniseg (#2683), urllib3, wx.rc (#2295),
  - numpy: Look for .dylib libraries, too ( #2544), support numpy MKL builds (#1881, #2111)
  - osgeo: Add conda specific places to check for auxiliary data (#2401)
  - QT and related
    - Add hooks for PySide2
    - Eliminate run-time hook by placing files in the correct directory
    - Fix path in homebrew for searching for qmake (#2354)
    - Repair Qt dll location (#2403)
    - Bundle PyQt 5.7 DLLs (#2152)
    - PyQt5: Return qml plugin path including subdirectory (#2694)
    - Fix hooks for PyQt5.QtQuick (#2743)
    - PyQt5.QtWebEngineWidgets: Include files needed by QWebEngine
  - GKT+ and related
    - Fix Gir file path on windows.
    - Fix unnecessary file search & generation when GI’s typelib is exists
    - gi: change gir search path when running from a virtualenv
    - gi: package gdk-pixbuf in osx codesign agnostic dir
    - gi: rewrite the GdkPixbuf loader cache at runtime on Linux
    - gi: support onefile mode for GdkPixbuf
    - gi: support using gdk-pixbuf-query-loaders-64 when present
    - gi: GIR files are only required on OSX
    - gio: copy the mime.cache also
    - Fix hooks for PyGOoObject on windows platform (#2306)
- Fixed hooks: botocore (#2384), clr (#1801), gstreamer (#2417), h5py (#2686), pylint, Tix data files (#1660), usb.core (#2088), win32com on non-windows-systems (#2479)
- Fix `multiprocess` spawn mode on POSIX OSs (#2322, #2505, #2759, #2795).
Bootloader

- `tempdir` option to control where bootloader will extract files (#2221)
- (Windows) in releases posted on PyPI requires msvcr*.dll (#2343)
- Fix unsafe string manipulation, resource and memory leaks. Thanks to Vito Kortbeek (#2489, #2502, #2503)
- Remove a left-over use of `getenv()`
- Set proper LISTEN_PID (set by `systemd`) in child process (#2345)
- Adds PID to bootloader log messages (#2466, #2480)
- (Windows) Use `_wputenv_s()` instead of `SetEnvironmentVariableW()`
- (Windows) Enhance error messages (#1431)
- (Windows) Add workaround for a Python 3 issue http://bugs.python.org/issue29778 (#2496, #2844)
- (OS X): Use single process for –onedir mode (#2616, #2618)
- (GNU/Linux) Compile bootloaders with –no-lsb by default (#2369)
- (GNU/Linux) Fix: linux64 bootloader requires glibc 2.14 (#2160)
- (GNU/Linux) `set_dynamic_library_path` change breaks plugin library use (#625)

Bootloader build

The bootloader build was largely overhauled. In the wscript, the build no longer depends on the Python interpreter’s bit-size, but on the compiler. We have a machine for building bootloaders for Windows and cross-building for OS X. Thus all maintainer are now able to build the bootloaders for all supported platforms.

- Add “official” build-script.
- (GNU/Linux) Make –no-lsb the default, add option –lsb.
- Largely overhauled Vagrantfile:
  - Make Darwin bootloaders build in OS X box (unused)
  - Make Windows bootloaders build using MSVC
  - Allow specifying cross-target on linux64.
  - Enable cross-building for OS X.
  - Enable cross-building for Windows (unused)
  - Add box for building osxcross.
- Largely overhauled wscript:
  - Remove options –target-cpu.
  - Use compiler’s target arch, not Python’s.
  - Major overhaul of the script
  - Build zlib if required, not if “on windows”.
  - Remove obsolete warnings.
  - Update Solaris, AIX and HPUX support.
  - Add flags for ‘strip’ tool in AIX platform.
Don’t set POSIX / SUS version defines.

- (GNU/Linux) for 64-bit arm/aarch ignore the gcc flag -m64 (#2801).

### Module loader

- Implement PEP-451 ModuleSpec type import system (#2377)
- Fix: Import not thread-safe? (#2010, #2371)

### PyInstaller Core

- Analyze: Check Python version when testing whether to rebuild.
- Analyze: Don’t fail on syntax error in modules, simply ignore them.
- Better error message when datas are not found. (#2308)
- Building: OSX: Use unicode literals when creating Info.plist XML.
- Building: Don’t fail if “datas” filename contain glob special characters. (#2314)
- Building: Read runtime-tmpdir from .spec-file.
- Building: Update a comment.
- building: warn users if bincache gets corrupted. (#2614)
- Cli-utils: Remove graceful handling of obsolete command line options.
- Configure: Create new parent-dir when moving old cache-dir. (#2679)
- Depend: Include vcruntime140.dll on Windows. (#2487)
- Depend: print nice error message if analyzed script has syntax error.
- Depend: When scanning for ctypes libs remove non-basename binaries.
- Enhance run-time error message on ctypes import error.
- Fix #2585: py2 non-unicode sys.path been tempted by os.path.abspath(). (#2585)
- Fix crash if extension module has hidden import to ctypes. (#2492)
- Fix handling of obsolete command line options. (#2411)
- Fix versioninfo.py breakage on Python 3.x (#2623)
- Fix: “Unicode-objects must be encoded before hashing” (#2124)
- Fix: UnicodeDecodeError - collect_data_files does not return filenames as unicode (#1604)
- Remove graceful handling of obsolete command line options. (#2413)
- Make grab version more polite on non-windows (#2054)
- Make utils/win32/versioninfo.py round trip the version info correctly.
- Makespec: Fix version number processing for PyCrypto. (#2476)
- Optimizations and refactoring to modulegraph and scanning for ctypes dependencies.
- pyinstaller should not crash when hitting an encoding error in source code (#2212)
- Remove destination for COLLECT and EXE prior to copying it (#2701)
- Remove uninformative traceback when adding not found data files (#2346)
• threading bug while processing imports (#2010)
• utils/hooks: Add logging to collect_data_files.
• (win32) Support using pypiwin32 or pywin32-ctypes (#2602)
• (win32) Use os.path.normpath to ensure that system libs are excluded.
• (win32) Look for libpython%.%_dll in Windows MSYS2 (#2571)
• (win32) Make versioninfo.py round trip the version info correctly (#2599)
• (win32) Ensure that pywin32 isn’t imported before check_requirements is called
• (win32) pyi-grab_version and –version-file not working? (#1347)
• (win32) Close PE() object to avoid mmap memory leak (#2026)
• (win32) Fix: ProductVersion in windows version info doesn’t show in some cases (#846)
• (win32) Fix multi-byte path bootloader import issue with python2 (#2585)
• (win32) Forward DYLD_LIBRARY_PATH through arch command. (#2035)
• (win32) Add vcruntime140.dll to win_includes for Python 3.5 an 3.6 (#2487)
• (OS X) Add libpython%d.%d.dylib to Darwin (is_darwin) PYDYLIB_NAMES. (#1971)
• (OS X) macOS bundle Info.plist should be in UTF-8 (#2615)
• (OS X) multiprocessing spawn in python 3 does not work on macOS (#2322)
• (OS X) Pyinstaller not able to find path (@rpath) of dynamic library (#1514)
• Modulegraph
  – Align with upstream version 0.13.
  – Add the upstream test-suite
  – Warn on syntax error and unicode error. (#2430)
  – Implement enumerate_instructions() (#2720)
  – Switch byte-code analysis to use Instruction (like dis3 does) (#2423)
  – Log warning on unicode error instead of only a debug message (#2418)
  – Use standard logging for messages. (#2433)
  – Fix to reimport failed SWIG C modules (1522, #2578).
• Included 3rd-party libraries
  – Remove bundled pefile and macholib, use the releases from PyPI. (#1920, #2689)
  – altgraph: Update to altgraph 0.13, add upstream test-suite.

Utilities

• grab_version.py: Display a friendly error message when utility fails (#859, #2792).
Test-suite and Continuous Integration

- Rearrange requirements files.
- Pin required versions – now updated using pyup (#2745)
- Hide useless trace-backs of helper-functions.
- Add a test for PyQt5.QtQuick.
- Add functional tests for PySide2
- Add test for new feature –runtime-tmpdir.
- Fix regression-test for #2492.
- unit: Add test-cases for PyiModuleGraph.
- unit/altgraph: Bringing in upstream altgraph test-suite.
- unit/modulegraph: Bringing in the modulegraph test-suite.
- Continuous Integration
  - Lots of enhancements to the CI tests to make them more stabile and reliable.
  - Pin required versions – now updated using pyup (#2745)
  - OS X is now tested along with GNU/Linux at Travis CI (#2508)
  - Travis: Use stages (#2753)
  - appveyor: Save cache on failure (#2690)
  - appveyor: Verify built bootloaders have the expected arch.

Documentation

- Add information how to donate (#2755, #2772).
- Add how to install the development version using pip.
- Fix installation instructions for development version. (#2761)
- Better examples for hidden imports.
- Clarify and fix “Adding Data Files” and “Adding Binary Files”. (#2482)
- Document new command line option ‘--runtime-tmpdir’.
- pyinstaller works on powerpc linux, big endian arch (#2000)
- Largely rewrite section “Building the Bootloader”, update from the wiki page.
- Describe building LSB-compliant bootloader as (now) special case.
- help2rst: Add cross-reference labels for option-headers.
- Enable sphinx.ext.intersphinx and links to our website.
- Sphinx should not “adjust” display of command line documentation (#2217)
Known Issues

- Data-files from wheels, unzipped eggs or not ad egg at all are not included automatically. This can be worked around using a hook-file, but may not suffice when using --onefile and something like python-daemon.
- The multipackage (MERGE) feature (#1527) is currently broken.
- (OSX) Support for OpenDocument events (#1309) is broken.
- (Windows) With Python 2.7 the frozen application may not run if the user-name (more specifically %TEMPDIR%) includes some Unicode characters. This does not happen with all Unicode characters, but only some and seems to be a windows bug. As a work-around please upgrade to Python 3 (#2754, #2767).
- (Windows) For Python >= 3.5 targeting Windows < 10, the developer needs to take special care to include the Visual C++ run-time .dlls. Please see the section Platform-specific Notes in the manual. (#1566)
- For Python 3.3, imports are not thread-safe (#2371#). Since Python 3.3 is end of live at 2017-09-29, we are not going to fix this.

1.14.5 Older Versions

Changelog for PyInstaller 3.0 – 3.2.1

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 msvcr10.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), gstreamer (#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 parameter 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 byte-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, PyExcelerate, PyObjcet, 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.

Changelog for PyInstaller 2.x

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, jinja2, 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 usefull 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: archieve_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 programatically.
• 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).
PyInstaller Documentation, Release 3.5

- (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.
- Add import hook for PyQt4.
- Add import hook for win32com.
- Improve support for running PyInstaller in virtualenv.
- 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 old platforms (dos, os2, MacOS 9).
- 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 in- stead.
- (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/

Changelog for PyInstaller 1.x

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.

1.14. Changelog for PyInstaller
• 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 Dwiggins)
PyInstaller Documentation, Release 3.5

- 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.

1.14. Changelog for PyInstaller
- 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 SQLObject 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 recompiled 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)

1.15 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:

1.15.1 Contributions to PyInstaller 3.5

• Hartmut Goebel - Core developer, maintainer and release manager.
• Bryan A. Jones - Core developer and PyQt5-tamer.

1.15.2 Contributions to PyInstaller 3.4

• Hartmut Goebel - Core developer, maintainer and release manager.
• Bryan A. Jones - Core developer and PyQt5-tamer.
• David Vierra - Core developer and encoding specialist.
• xoviat - brave contributor
• Hugo vk - brave contributor

1.15.3 Contributions to PyInstaller 3.3.1

• Hartmut Goebel - Core developer and release manager.
• Bryan A. Jones - Core developer.
• David Vierra - Core developer and encoding specialist.
• xoviat - brave contributor
1.15.4 Contributions to PyInstaller 3.3

Special Thanks xiovat for implementing Python3.6 support and to Jonathan Springer and xoviat for stabilizing the continuous integration tests.

- Hartmut Goebel - Core developer and release manager.
- Bryan A. Jones - Core developer.
- David Vierra - Core developer and encoding specialist.
- xoviat - brave programmer
- Jonathan Springer
- Vito Kortbeek
- Dustin Spicuzza
- Ben Hagen
- Paavo
- Brian Teague
- Chris Norman
- Jonathan Stewmon
- Guillaume Thiolliere
- Justin Harris
- Kenneth Zhao
- Paul Müller
- giumas
- y2kbugger

1.15.5 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 Akhiyarov
• 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
1.15.6 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 Schoorisse - 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.

1.15.7 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
1.15.8 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 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.

1.15.9 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 - PyExcelerate 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 Vrhovec - 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.

1.15.10 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 <email 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 Hoech - 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 Dwiggins - pyodbc import hook.
• Allan Green - refactoring and improved in-process COM servers.
• Daniele Varrazzo - various bootloader and OS X fixes.
• Greg Copeland - sqlalchehy import hook.
• Seth Remington - PyGTK hook improvements.
• Marco Bonifazi - PyGTK hook improvements. PyOpenGL import hook.
• Jamie Kirkpatrick - paste import hook.
PyInstaller Documentation, Release 3.5

• 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.

1.16 Man Pages

1.16.1 pyinstaller

SYNOPSIS

pyinstaller <options> SCRIPT...
pyinstaller <options> SPECFILE

DESCRIPTION

PyInstaller is a program that freezes (packages) Python programs into stand-alone executables, under Windows, GNU/Linux, Mac OS X, FreeBSD, Solaris and AIX. Its main advantages over similar tools are that PyInstaller works with Python 2.7 and 3.4—3.6, it builds smaller executables thanks to transparent compression, it is fully multi-platform, and use the OS support to load the dynamic libraries, thus ensuring full compatibility.

You may either pass one or more file-names of Python scripts or a single .spec-file-name. In the first case, pyinstaller will generate a .spec-file (as pyi-makespec would do) and immediately process it.

If you pass a .spec-file, this will be processed and most options given on the command-line will have no effect. 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: ./dist)
--workpath WORKPATH Where to put all the temporary work files, .log, .pyz and etc. (default: ./build)
-y, --noconfirm Replace output directory (default: SPECPATH/dist/SPECNAME) 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 <all,imports,bootloader,noarchive>, --debug <all,imports,bootloader,noarchive>  Provide assistance with debugging a frozen application. This argument may be provided multiple times to select several of the following options.

  • all: All three of the following options.
  • imports: specify the -v option to the underlying Python interpreter, causing it to print a message each time a module is initialized, showing the place (filename or built-in module) from which it is loaded. See https://docs.python.org/3/using/cmdline.html#id4.
• bootloader: tell the bootloader to issue progress messages while initializing and starting the bundled app. Used to diagnose problems with missing imports.

• noarchive: instead of storing all frozen Python source files as an archive inside the resulting executable, store them as files in the resulting output directory.

-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)

--upx-exclude FILE Prevent a binary from being compressed when using upx. This is typically used if upx corrupts certain binaries during compression. FILE is the filename of the binary without path. This option can be used multiple times.

Windows and Mac OS X specific options

-c, --console, --nowindowed Open a console window for standard i/o (default). On Windows this option will have no effect if the first script is a '.pyw' file.

-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. On Windows this option will be set if the first script is a '.pyw' file. 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 *. add 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)

Rarely used special options

--runtime-tmpdir PATH Where to extract libraries and support files in onefile-mode. If this option is given, the bootloader will ignore any temp-folder location defined by the runtime OS. The _MEIxxxxxx-folder will be created here. Please use this option only if you know what you are doing.

--bootloader-ignore-signals Tell the bootloader to ignore signals rather than forwarding them to the child process. Useful in situations where e.g. a supervisor process signals both the bootloader and child (e.g. via a process group) to avoid signalling the child twice.

ENVIRONMENT VARIABLES

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

SEE ALSO


1.16.2 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 <all,imports,bootloader,noarchive>, --debug <all,imports,bootloader,noarchive>  Provide assistance with debugging a frozen application. This argument may be provided multiple times to select several of the following options. - all: All three of the following options. - imports: specify the -v option to the underlying Python interpreter, causing it to print a message each time a module is initialized, showing the place (filename or built-in module) from which it is loaded. See https://docs.python.org/3/using/cmdline.html#id4. - bootloader: tell the bootloader to issue progress messages while initializing and starting the bundled app. Used to diagnose problems with missing imports. - noarchive: instead of storing all frozen Python source files as an archive inside the resulting executable, store them as files in the resulting output directory.

-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)

--upx-exclude FILE  Prevent a binary from being compressed when using upx. This is typically used if upx corrupts certain binaries during compression. FILE is the filename of the binary without path. This option can be used multiple times.

Windows and Mac OS X specific options

-c, --console, --nowindowed  Open a console window for standard i/o (default). On Windows this option will have no effect if the first script is a `.pyw` file.

-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. On Windows this option will be set if the first script is a `.pyw` file. 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 LAN-
GUAGE 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 al-
ways 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 appli-
cation, PyInstaller will prefer not to follow policies that redirect to newer ver-
sions, 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)

Rarely used special options

--runtime-tmpdir PATH Where to extract libraries and support files in onefile-mode. If this option 
given, the bootloader will ignore any temp-folder location defined by the run-
time OS. The _MEIxxxxxxx-folder will be created here. Please use this option 
only if you know what you are doing.

--bootloader-ignore-signals Tell the bootloader to ignore signals rather than forwarding them to the 
child process. Useful in situations where e.g. a supervisor process signals both 
the bootloader and child (e.g. via a process group) to avoid signalling the child 
twice.

ENVIRONMENT VARIABLES

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

SEE ALSO

pyinstaller.org
1.17 Development Guide

1.17.1 Quick-start

• Our git repository is at https://github.com/pyinstaller/pyinstaller:

  ```
git clone https://github.com/pyinstaller/pyinstaller
  ```

  – Development is done on the `develop` branch. Pull-request shall be filed against this branch.
  – Releases will reside on the `master` branch.

• Install required testing tools:

  ```
pip install -r tests/requirements-tools.txt
  ```

• Commit as often as you'd like, but squash or otherwise rewrite your commits into logical patches before asking for code review. `git rebase -i` is your friend. Read the » Detailed Commit Guideline for more information.

  Reformating code without functional changes will generally not be accepted (for rational see #2727).

• Write meaningful commit messages.

  – The first line shall be a short sentence that can stand alone as a short description of the change, written in the present tense, and prefixed with the `subsystem-name`.

  – The body of the commit message should explain or justify the change. Read the » Detailed Commit Message Rules for more information.

• Provide tests that cover your changes and try to run the tests locally first.

• Submit pull-requests against the `develop` branch. Mind adding a `changelog entry` so our users can learn about your change!

• For new files mind adding the copyright header, see `PyInstaller/__init__.py` (also mind updating to the current year).

• In response to feedback, squash the new “fix up” commits into the respective commit that is being fixed with an interactive rebase (`git rebase -i`). *Push the new, rewritten branch with a `git push --force`. (Scary! But github doesn’t play nicely with a safer method.)*

New to GitHub or Git?

Our development workflow is build around Git and GitHub. Please take your time to become familiar with these. If you are new to GitHub, GitHub has instructions for getting you started. If you are new to Git there are a tutorial and an excellent book available online.

Further Reading

• Please Write Good Commit Messages

• Creating Pull-Requests

• Updating a Pull-Request

• PyInstaller’s Branch Model
Coding conventions

The PyInstaller project follows the PEP 8 Style Guide for Python Code for new code. Please check your code with a style guide checker, e.g. flake8.

Please abstain from reformatting existing code, even if it doesn’t follow PEP 8. We will not accept reformatting changes since they make it harder to review the changes and to follow changes in the long run. For a complete rationale please see #2727.

Running the Test Suite

To run the test-suite, please proceed as follows.

1. If you don’t have a git clone of PyInstaller, first fetch the current development head, either using pip,...:
   
   ```bash
   → zip
   unzip develop.zip
   cd pyinstaller-develop/
   ...
   ```

   or using git:

   ```bash
   git clone https://github.com/pyinstaller/pyinstaller.git
   cd pyinstaller
   ```

2. Then setup a fresh virtualenv for running the test suite in and install all required tools:

   ```bash
   pip install --user virtualenv
   virtualenv /tmp/venv
   . /tmp/venv/bin/activate
   pip install -r tests/requirements-tools.txt
   ```

3. To run a single test use e.g.:

   ```bash
   py.test tests/unit -k test_collect_submod_all_included
   ```

4. Run the test-suite:

   ```bash
   py.test tests/unit tests/functional
   ```

   This only runs the tests for the core functionality and some packages from the Python standard library.

5. To get better coverage, including many of the available hooks, you need to download the Python packages to be tested. For this please run:

   ```bash
   pip install -U -r tests/requirements-libraries.txt
   py.test tests/unit tests/functional
   ```

To learn how we run the test-suite in the continuous integration tests please have a look at .travis.yml (for GNU/Linux and OS X) and appveyor.yml (for Windows).

Guidelines for Commits

Please help keeping code and changes comprehensible for years. Provide a readable commit-history following this guideline.

A commit
• stands alone as a single, complete, logical change,
• has a descriptive commit message (see below),
• has no extraneous modifications (whitespace changes, fixing a typo in an unrelated file, etc.),
• follows established coding conventions (PEP 8) closely.

Avoid committing several unrelated changes in one go. It makes merging difficult, and also makes it harder to determine which change is the culprit if a bug crops up.

If you did several unrelated changes before committing, `git gui` makes committing selected parts and even selected lines easy. Try the context menu within the windows diff area.

This results in a more readable history, which makes it easier to understand why a change was made. In case of an issue, it’s easier to `git bisect` to find breaking changes any revert those breaking changes.

**In Detail**

A commit should be one (and just one) logical unit. It should be something that someone might want to patch or revert in its entirety, and never piece-wise. If it could be useful in pieces, make separate commits.

• Make small patches (i.e. work in consistent increments).
• Reformating code without functional changes will generally not be accepted (for rationale see #2727). If such changes are required, separate it into a commit of its own and document as such.

This means that when looking at patches later, we don’t have to wade through loads of non-functional changes to get to the relevant parts of the patch.

• Especially don’t mix different types of change, and put a standard prefix for each type of change to identify it in your commit message.
• Abstain refactorings! If any, restrict refactorings (that should not change functionality) to their own commit (and document).
• Restrict functionality changes (bug fix or new feature) to their own changelists (and document).
• If your commit-series includes any “fix up” commits (“Fix typo.”, “Fix test.”, “Remove commented code.”) please use `git rebase -i ...` to clean them up prior to submitting a pull-request.

Please Write Good Commit Messages

Please help keeping code and changes comprehensible for years. Write good commit messages following this guideline.

Commit messages should provide enough information to enable a third party to decide if the change is relevant to them and if they need to read the change itself.

*PyInstaller* is maintained since 2005 and we often need to comprehend years later why a certain change has been implemented as it is. What seemed to be obvious when the change was applied may be just obscure years later. The original contributor may be out of reach, while another developer needs to comprehend the reasons, side-effects and decisions the original author considered.

We learned that commit messages are important to comprehend changes and thus we are a bit picky about them.

We may ask you to reword your commit messages. In this case, use `git rebase -i ...` and `git push -f ...` to update your pull-request. See *Updating a Pull-Request* for details.
Content of the commit message

Write meaningful commit messages.

- The first line shall be a short sentence that can stand alone as a short description of the change, written in the present tense, and prefixed with the subsystem-name. See below for details.
- The body of the commit message should explain or justify the change, see below for details.

Examples of good commit messages are 5c1628e6 or 73d77106.

The first Line

The first line of the commit message shall

- be a short sentence (72 characters maximum, but shoot for 50),
- use the present tense (“Add awesome feature.”)¹,
- be prefixed with an identifier for the subsystem this commit is related to (“tests: Fix the frob.” or “building: Make all nodes turn faster.”),
- always end with a period.
- Ending punctuation other than a period should be used to indicate that the summary line is incomplete and continues after the separator; “…” is conventional.

The Commit-Message Body

The body of a commit log should:

- explain or justify the change,
  - If you find yourself describing implementation details, this most probably should go into a source code comment.
  - Please include motivation for the change, and contrasts its implementation with previous behavior.
  - For more complicate or serious changes please document relevant decisions, contrast them with other possibilities for chosen, side-effect you experienced, or other thinks to keep in mind when touching this peace of code again. (Although the later might better go into a source code comment.)
- for a bug fix, provide a ticket number or link to the ticket,
- explain what changes were made at a high level (The GNU ChangeLog standard is worth a read),
- be word-wrapped to 72 characters per line, don’t go over 80; and
- separated by a blank line from the first line.

- Bullet points and numbered lists are okay, too:

  * Typically a hyphen or asterisk is used for the bullet, preceded by a single space, with blank lines in between, but conventions vary here.
  * Use a hanging indent.

¹ Consider these messages as the instructions for what applying the commit will do. Further this convention matches up with commit messages generated by commands like git merge and git revert.
• Do not start your commit message with a hash-mark (#) as git some git commands may dismiss these message.  
  (See this discussion. for details.)

**Standard prefixes**

Please state the “subsystem” this commit is related to as a prefix in the first line. Do learn which prefixes others used for the files you changed you can use `git log --oneline path/to/file/or/dir`

Examples for “subsystems” are:

• Hooks for hook-related changes
• Bootloader, Bootloader build for the bootloader or it’s build system
• depend for the dependency detection parts (PyInstaller/depend)
• building for the building part (PyInstaller/building)
• compat for code related to compatibility of different Python versions (primary PyInstaller/compat.py)
• loader
• utils,utils/hooks
• Tests, Test/CI: For changes to the test suite (incl. requirements), resp. the CI.
• modulegraph: changes related to PyInstaller/lib/modulegraph
• Doc, Doc build for the documentation content resp. it’s build system. You may want to specify the chapter or section too.

**Please set the correct Author**

Please make sure you have setup git to use the correct name and email for your commits. Use the same name and email on all machines you may push from. Example:

```bash
# Set name and email
git config --global user.name "Firstname Lastname"
git config --global user.email "your_email@youremail.com"
```

This will set this name and email-address to be used for all git-repos you are working on on this system. To set it for just the PyInstaller repo, remove the --global flag.

Alternatively you may use `git gui → Edit → Options ...` to set these values.

**Further Reading**

Further hints and tutorials about writing good commit messages can also be found at:

• FreeBSD Committer’s Guide
• http://365git.tumblr.com/post/3308646748/writing-git-commit-messages
• http://wincent.com/blog/commit-messages: The Good, the Bad and the Ugly.
• http://wiki.scummvm.org/index.php/Commit_Guidelines
• http://lbrandy.com/blog/2009/03/writing-better-commit-messages/
• http://blog.looplabel.net/2008/07/28/best-practices-for-version-control/
PyInstaller Documentation, Release 3.5

- http://subversion.apache.org/docs/community-guide/conventions.html (Targeted a bit too much to subversion usage, which does not use such fine-grained commits as we ask you strongly to use.)

Credits

This page was composed from material found at
- http://hackage.haskell.org/trac/ghc/wiki/WorkingConventions/Git
- http://lbrandy.com/blog/2009/03/writing-better-commit-messages/
- https://git.dthompson.us/presentations.git/tree/HEAD:/happy-patching
- and other places.

Improving and Building the Documentation

PyInstaller’s documentation is created using Sphinx. Sphinx uses reStructuredText as its markup language, and many of its strengths come from the power and straightforwardness of reStructuredText and its parsing and translating suite, the Docutils.

The documentation is maintained in the Git repository along with the code and pushing to the develop branch will create a new version at https://pyinstaller.readthedocs.io/en/latest/.

For small changes (like typos) you may just fork PyInstaller on Github, edit the documentation online and create a pull-request.

For anything else we ask you to clone the repository and verify your changes like this:

```
pip install sphinx sphinx_rtd_theme
cd doc
make html
xdg-open _build/html/index.html
```

Please watch out for any warning and error while building the documentation. In your browser check if the markup is Prior to pushing your changes and creating the pull-request please run:

```
make clean html
```

to verify once again everything is fine. Thank you!

We may ask you to rework your changes or reword your commit messages. In this case, use git rebase -i ... and git push -f ... to update your pull-request. See Updating a Pull-Request for details.

PyInstaller extensions

For the PyInstaller documentation there are roles available* in additon to the ones from Sphinx and docutils.

**:commit:**

Refer to a commit, creating a web-link to the online git repository. The commit-id will be shortened to 8 digits for readability. Example: :commit:`a1b2c3d4e5f6a7b8c9` will become a1b2c3d4.

* Defined in `doc/_extensions/pyi_sphinx_roles.py`
:issue:

Link to an issue or pull-request number at Github. Example: :issue:`123` will become #123.

**reStructuredText Cheat-sheet**

- Combining markup and links:

  ```restructuredtext
  The easiest way to install PyInstaller is using |pip|_:
  
  .. |pip| replace:: :command:`pip`
  .. _pip: https://pip.pypa.io/
  ```

**Creating Pull-Requests**

**Example**

- Create an account on [https://github.com](https://github.com)
- Create a fork of project pyinstaller/pyinstaller on github.
- Set up your git client by following [this documentation on github](https://github.com).
- Clone your fork to your local machine:

  ```bash
git clone git@github.com:YOUR_GITHUB_USERNAME/pyinstaller.git
cd pyinstaller
  ```

- Develop your changes (aka “hack”)
  - Create a branch to work on (optional):

    ```bash
    git checkout -b my-patch
    ```

  - If you are going to implement a hook, start with creating a minimalistic build-test (see below). You will need to test your hook anyway, so why not use a build-test from the start?

  - Incorporate your changes into PyInstaller.

  - Test your changes by running all build tests to ensure nothing else is broken. Please test on as many platform as you can.

  - You may reference relevant issues in commit messages (like #1259) to make GitHub link issues and commits together, and with phrase like “fixes #1259” you can even close relevant issues automatically.

- Synchronize your fork with the PyInstaller upstream repository. There are two ways for this:

  1. Rebase you changes on the current development head (preferred, as it results in a straighter history and conflicts are easier to solve):

     ```bash
     git remote add upstream https://github.com/pyinstaller/pyinstaller.git
     git checkout my-patch
     git pull --rebase upstream develop
     git log --online --graph
     ```

  2. Merge the current development head into your changes:
git remote add upstream https://github.com/pyinstaller/pyinstaller.git
git fetch upstream develop
git checkout my-patch
git merge upstream/develop

git log --online --graph

For details see syncing a fork at github.

• Push your changes up to your fork:
  
git push

• Open the Pull Requests page at https://github.com/YOUR_GITHUB_USERNAME/pyinstaller/pulls and click “New pull request”. That’s it.

Updating a Pull-Request

We may ask you to update your pull-request to improve it’s quality or for other reasons. In this case, use git rebase -i ... and git push -f ... as explained below.¹ Please do not close the pull-request and open a new one – this would kill the discussion thread.

This is the workflow without actually changing the base:

```bash
git checkout my-branch
# find the commit your branch forked from 'develop'
mb=$(git merge-base --fork-point develop)
# rebase interactively without actually changing the base
git rebase -i $mb
# ... process rebase
git push -f my-fork my-branch
```

Or if you want to actually base your code on the current development head:

```bash
git checkout my-branch
# rebase interactively on 'develop'
git rebase -i develop
# ... process rebase
git push -f my-fork my-branch
```

Changelog Entries

If your change is noteworthy, there needs to be a changelog entry so our users can learn about it!

To avoid merge conflicts, we use the towncrier package to manage our changelog. towncrier uses independent files for each pull request – called news fragments – instead of one monolithic changelog file. On release, those news fragments are compiled into our doc/CHANGELOG.rst.

You don’t need to install towncrier yourself, you just have to abide by a few simple rules:

- For each pull request, add a new file into news/ with a filename adhering to the pr#. (feature|bugfix|breaking).rst schema: For example, news/42.feature.rst for a new feature that is proposed in pull request #42.

¹ There are other ways to update a pull-request, e.g. by “amending” a commit. But for casual (and not-so-casual :-) users rebase -i might be the easiest way.
Our categories are: feature, bugfix, break (breaking changes), hooks (all hook-related changes), bootloader, moduleloader, doc, process (project infrastructure, development process, etc.), core, build (the bootloader build process), and tests.

- As with other docs, please use semantic newlines within news fragments.
- Prefer present tense or constructions with “now” or “new”. For example:
  - Add hook for my-fancy-library.
  - Fix crash when trying to add resources to Windows executable using --resource option.
If the change is relevant only for a specific platform, use a prefix, like here:
  - (GNU/Linux) When building with --debug turn off FORTIFY_SOURCE to ease debugging.
- Wrap symbols like modules, functions, or classes into double backticks so they are rendered in a monospace font. If you mention functions or other callables, add parentheses at the end of their names: `is_module()`.
This makes the changelog a lot more readable.
- If you want to reference multiple issues, copy the news fragment to another filename. `towncrier` will merge all news fragments with identical contents into one entry with multiple links to the respective pull requests. You may also reference to an existing newsfragment by copying that one.
- If your pull-request includes several distinct topics, you may want to add several news fragment files. For example `4242.feature.rst` for the new feature, `4242.bootloader` for the accompanying change to the bootloader.

Remember that a news entry is meant for end users and should only contain details relevant to an end user.

**pyenv and PyInstaller**

**Note:** This section is still a draft. Please help extending it.

- clone pyenv repository:
  ```
  git clone https://github.com/yyuu/pyenv.git ~/.pyenv
  ```
- clone virtualenv plugin:
  ```
  ```
- add to `.bashrc` or `.zshrc`:
  ```
  # Add 'pyenv' to PATH.
  export PYENV_ROOT="$HOME/.pyenv"
  export PATH="$PYENV_ROOT/bin:$PATH"

  # Enable shims and autocompletion for pyenv.
  eval "$pyenv init -"
  # Load pyenv-virtualenv automatically by adding
  # the following to ~/.zshrc:
  
  eval "$pyenv virtualenv-init -"
  ```
- Install python version with shared libpython (necessary for Pyl installer to work):
env PYTHON_CONFIGURE_OPTS="--enable-shared" pyenv install 3.5.0

- setup virtualenv pyenv virtualenv 3.5.0 venvname
- activate virtualenv pyenv activate venvname
- deactivate virtualenv pyenv deactivate

**PyInstaller’s Branch Model**

*develop branch*  We consider *origin/develop* to be the main branch where the source code of HEAD always reflects a state with the latest delivered development changes for the next release. Some would call this the “integration branch”.

*master branch*  We consider *origin/master* to be the main branch where the source code of HEAD always reflects a *production-ready* state. Each commit to master is considered a new release and will be tagged.

The *PyInstaller* project doesn’t use long living branches (beside *master* and *develop*) as we don’t support bugfixes for several major releases in parallel.

Occasionally you might find these branches in the repository:

*release/* branches These branches are for preparing the next release. This is for example: updating the version numbers, completing the change-log, recompiling the bootloader, rebuilding the manuals. See ref:*release-workflow* for details about the release process and what steps have to be performed.

*hotfix/* branches These branches are also meant to prepare for a new production release, albeit unplanned. This is what is commonly known as a “hotfix”.

*feature/* branches Feature branches (or sometimes called topic branches) are used to develop new features for the upcoming or a distant future release.

### 1.18 Indices and tables

- genindex
- modindex
- search

---

1 This branching-model is basically the same as Vincent Driessen described in this blog. But currently we are not following it strictly.
C
commit (role), 102

I
issue (role), 102

P
Python Enhancement Proposals
PEP 302, 38
PEP 405, 17
PEP 527, 67
PEP 8, 98, 99