PyInstaller Documentation

Release 6.6.0+gca1ec5b

David Cortesi

2024-04-16

CONTENTS

1 Quickstart **3**

2 Contents: **5**

2.1 Requirements . . . 5
2.2 License . . . 6
2.3 How To Contribute . . . 6
2.4 How to Install PyInstaller . . . 7
2.5 What PyInstaller Does and How It Does It . . . 9
2.6 Using PyInstaller . . . 12
2.7 Common Issues and Pitfalls . . . 26
2.8 Run-time Information . . . 33
2.9 Using Spec Files . . . 36
2.10 Notes about specific Features . . . 49
2.11 When Things Go Wrong . . . 74
2.12 Advanced Topics . . . 78
2.13 Understanding PyInstaller Hooks . . . 87
2.14 Hook Configuration Options . . . 105
2.15 Building the Bootloader . . . 112
2.16 Changelog for PyInstaller . . . 119
2.17 Credits . . . 195
2.18 Man Pages . . . 211
2.19 Development Guide . . . 220
2.20 Indices and tables . . . 230

Python Module Index **231**

Index **233**

Version PyInstaller 6.6.0+gca1ec5b

Homepage https://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 3.8 and newer, and correctly bundles many major Python packages such as numpy, matplotlib, PyQt, wxPython, and others.

PyInstaller is tested against Windows, MacOS X, and Linux. However, it is not a cross-compiler; to make a Windows app you run PyInstaller on Windows, and to make a Linux app you run it on Linux, etc. x PyInstaller has been used successfully with AIX, Solaris, FreeBSD and OpenBSD but testing against them is not part of our continuous integration tests, and the development team offers no guarantee (all code for these platforms comes from external contributions) that PyInstaller will work on these platforms or that they will continue to be supported.

CHAPTER

ONE

QUICKSTART

Make sure you have the *Requirements* installed, and then install PyInstaller from PyPI:

```
pip install -U pyinstaller
```

Open a command prompt/shell window, and navigate to the directory where your *.py* file is located, then build your app with the following command:

```
pyinstaller your_program.py
```

Your bundled application should now be available in the *dist* folder.

CHAPTER

TWO

CONTENTS:

2.1 Requirements

2.1.1 Windows

PyInstaller runs in Windows 8 and newer. It can create graphical windowed apps (apps that do not need a command window).

2.1.2 macOS

PyInstaller runs on macOS 10.15 (Catalina) or newer. It can build graphical windowed apps (apps that do not use a terminal window). PyInstaller builds apps that are compatible with the macOS release in which you run it, and following releases. It can build `x86_64`, `arm64` or hybrid *universal2* binaries on macOS machines of either architecture. See *macOS multi-arch support* for details.

2.1.3 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`.

2.1.4 AIX, Solaris, FreeBSD and OpenBSD

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*.

2.2 License

PyInstaller is distributed under a dual-licensing scheme using both the GPL 2.0 License, with an exception that allows you to use it to build commercial products - listed below - and the Apache License, version 2.0, which only applies to a certain few files. To see which files the Apache license applies to, and to which the GPL applies, please see the *COPYING.txt* file which can be found in the root of the PyInstaller source repository.

A quick summary of the GPL license exceptions:

- **You may use PyInstaller to bundle commercial applications out of your** source code.
- **The executable bundles generated by PyInstaller from your source code** can be shipped with whatever license you want, as long as it complies with the licenses of your dependencies.
- **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.

2.3 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 create a ticket on GitHub.

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

2.3.1 Some ideas how you can help

Some ideas how you can help:

- **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!

2.4 How to Install PyInstaller

PyInstaller is available as a regular Python package. The source archives for released versions are available from PyPi, but it is easier to install the latest version using pip:

```
pip install pyinstaller
```

To upgrade existing PyInstaller installation to the latest version, use:

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

To install the current development version, use:

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

To install directly using pip's built-in git checkout support, use:

```
pip install git+https://github.com/pyinstaller/pyinstaller
```

or to install specific branch (e.g., `develop`):

```
pip install git+https://github.com/pyinstaller/pyinstaller@develop
```

2.4.1 Installing from the source archive

The source code archive for released versions of PyInstaller are available at PyPI and on PyInstaller Downloads page.

Note: Even though the source archive provides the `setup.py` script, installation via `python setup.py install` has been deprecated and should not be used anymore. Instead, run `pip install .` from the unpacked source directory, as described below.

The installation procedure is:

1. Unpack the source archive.
2. Move into the unpacked source directory.
3. Run `pip install .` from the unpacked source directory. If installing into system-wide python installation, administrator privilege is required.

The same procedure applies to installing from manual git checkout:

```
git clone https://github.com/pyinstaller/pyinstaller
cd pyinstaller
pip install .
```

If you intend to make changes to the source code and want them to take effect immediately, without re-installing the package each time, you can install it in editable mode:

```
pip install -e .
```

For platforms other than Windows, GNU/Linux and macOS, you must first build the bootloader for your platform: see *Building the Bootloader*. After the bootloader has been built, use the `pip install .` command to complete the installation.

2.4.2 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 `4.n` for a released version, and `4.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:\Python38\Scripts` for Python 3.8)
- GNU/Linux: `/usr/bin/`
- macOS (using the default Apple-supplied Python) `/usr/bin`
- macOS (using Python installed by homebrew) `/usr/local/bin`
- macOS (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`.

Note: If you cannot use the `pyinstaller` command due to the scripts directory not being in `PATH`, you can instead invoke the `PyInstaller` module, by running `python -m PyInstaller` (pay attention to the module name, which is case sensitive). This form of invocation is also useful when you have PyInstaller installed in multiple python environments, and you cannot be sure from which installation the `pyinstaller` command will be ran.

2.4.3 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*.
- `pyi-set_version` can be used to apply previously-extracted version resource to an existing Windows executable.

2.5 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,

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

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

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

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

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

2.5.1 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, matplotlib, 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 `importlib.import_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.

2.5.2 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.)

2.5.3 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, macOS, 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.)

2.5.4 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.

2.5.5 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 *xxxxxx* is a random number.

The one executable file contains an embedded archive of all the Python modules used by your script, as well as compressed copies of any non-Python support files (e.g. `.so` files). The bootloader uncompresses the support files and writes copies into the 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.

Note: PyInstaller currently does not preserve file attributes. see #3926.

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. Alternatively, you can set the environment variable that controls the temporary directory before launching the program. See *Defining the Extraction Location*).

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 macOS). 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.

Note: Do *not* give administrator privileges to a one-file executable on Windows ("Run this program as an administrator"). There is an unlikely but not impossible way in which a malicious attacker could corrupt one of the shared libraries in the temp folder while the bootloader is preparing it. When distributing a privileged program in general, ensure that file permissions prevent shared libraries or executables from being tampered with. Otherwise, an unelevated process which has write access to these files may escalate privileges by modifying them.

Note: 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.

2.5.6 Using a Console Window

By default the bootloader creates a command-line console (a terminal window in GNU/Linux and macOS, 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 macOS 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 extension 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.

2.5.7 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.

2.6 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"
```

2.6.1 Options

A full list of the `pyinstaller` command's options are as follows:

Positional Arguments

scriptname
Name of scriptfiles to be processed or exactly one .spec file. If a .spec file is specified, most options are unnecessary and are ignored.

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)

--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, DEPRECATION, ERROR, FATAL (default: INFO). Also settable via and overrides the PYI_LOG_LEVEL environment variable.

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)

--contents-directory CONTENTS_DIRECTORY
For onedir builds only, specify the name of the directory in which all supporting files (i.e. everything except the executable itself) will be placed in. Use "." to re-enable old onedir layout without contents directory.

What To Bundle, Where To Search

--add-data SOURCE:DEST
: Additional data files or directories containing data files to be added to the application. The argument value should be in form of "source:dest_dir", where source is the path to file (or directory) to be collected, dest_dir is the destination directory relative to the top-level application directory, and both paths are separated by a colon (:). To put a file in the top-level application directory, use . as a dest_dir. This option can be used multiple times.

--add-binary SOURCE:DEST
: Additional binary files to be added to the executable. See the `--add-data` option for the format. 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. Equivalent to supplying the `pathex` argument in the spec file.

--hidden-import MODULENAME, **--hiddenimport** MODULENAME
: Name an import not visible in the code of the script(s). This option can be used multiple times.

--collect-submodules MODULENAME
: Collect all submodules from the specified package or module. This option can be used multiple times.

--collect-data MODULENAME, **--collect-datas** MODULENAME
: Collect all data from the specified package or module. This option can be used multiple times.

--collect-binaries MODULENAME
: Collect all binaries from the specified package or module. This option can be used multiple times.

--collect-all MODULENAME
: Collect all submodules, data files, and binaries from the specified package or module. This option can be used multiple times.

--copy-metadata PACKAGENAME
: Copy metadata for the specified package. This option can be used multiple times.

--recursive-copy-metadata PACKAGENAME
: Copy metadata for the specified package and all its dependencies. 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.

--splash IMAGE_FILE
: (EXPERIMENTAL) Add an splash screen with the image IMAGE_FILE to the application. The splash screen can display progress updates while unpacking.

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.

--optimize LEVEL
Bytecode optimization level used for collected python modules and scripts. For details, see the section "Bytecode Optimization Level" in PyInstaller manual.

--python-option PYTHON_OPTION
Specify a command-line option to pass to the Python interpreter at runtime. Currently supports "v" (equivalent to "–debug imports"), "u", "W <warning control>", "X <xoption>", and "hash_seed=<value>". For details, see the section "Specifying Python Interpreter Options" in PyInstaller manual.

-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 has 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 this also triggers building a Mac OS .app bundle. On Windows this option is automatically set if the first script is a '.pyw' file. This option is ignored on *NIX systems.

--hide-console {minimize-late,minimize-early,hide-early,hide-late}
Windows only: in console-enabled executable, have bootloader automatically hide or minimize the console window if the program owns the console window (i.e., was not launched from an existing console window).

-i <FILE.ico or FILE.exe,ID or FILE.icns or Image or "NONE">, **--icon** <FILE.ico or FILE.exe,ID or FILE.icns or Image or "NONE">
FILE.ico: apply the 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. If an image file is entered that isn't in the platform format (ico on Windows, icns on Mac), PyInstaller tries to use Pillow to translate the icon into the correct format (if Pillow is installed). Use "NONE" to not apply any icon, thereby making the OS show some default (default: apply PyInstaller's icon). This option can be used multiple times.

--disable-windowed-traceback
Disable traceback dump of unhandled exception in windowed (noconsole) mode (Windows and macOS only), and instead display a message that this feature is disabled.

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 that will request elevation upon application start.

--uac-uiaccess
: Using this option allows an elevated application to work with Remote Desktop.

Mac Os Specific Options

--argv-emulation
: Enable argv emulation for macOS app bundles. If enabled, the initial open document/URL event is processed by the bootloader and the passed file paths or URLs are appended to sys.argv.

--osx-bundle-identifier BUNDLE_IDENTIFIER
: Mac OS .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)

--target-architecture ARCH, **--target-arch** ARCH
: Target architecture (macOS only; valid values: x86_64, arm64, universal2). Enables switching between universal2 and single-arch version of frozen application (provided python installation supports the target architecture). If not target architecture is not specified, the current running architecture is targeted.

--codesign-identity IDENTITY
: Code signing identity (macOS only). Use the provided identity to sign collected binaries and generated executable. If signing identity is not provided, ad- hoc signing is performed instead.

--osx-entitlements-file FILENAME
: Entitlements file to use when code-signing the collected binaries (macOS only).

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 run-time OS. The `_MEIxxxxxx`-folder will be created here. Please use this option only if you know what you are doing. Note that on POSIX systems, PyInstaller's bootloader does NOT perform shell-style environment variable expansion on the given path string. Therefore, using environment variables (e.g., ~ or `$HOME`) in path will NOT work.

--bootloader-ignore-signals
: Tell the bootloader to ignore signals rather than forwarding them to the child process. Useful in situations where

for example a supervisor process signals both the bootloader and the child (e.g., via a process group) to avoid signalling the child twice.

2.6.2 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="image1.png:img" \
    --add-binary="libfoo.so:lib" \
    --hidden-import=secret1 \
    --hidden-import=secret2 \
    --upx-dir=/usr/local/share/ \
    myscript.spec
```

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

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

2.6.3 Running PyInstaller from Python code

If you want to run PyInstaller from Python code, you can use the `run` function defined in `PyInstaller.__main__`. For instance, the following code:

```
import PyInstaller.__main__

PyInstaller.__main__.run([
    'my_script.py',
    '--onefile',
    '--windowed'
])
```

Is equivalent to:

```
pyinstaller my_script.py --onefile --windowed
```

2.6.4 Using UPX

UPX is a free utility for compressing executable files and libraries. It is available for most operating systems and can compress a large number of executable file formats. See the UPX home page for downloads, and for the list of supported file formats.

When UPX is available, PyInstaller uses it to individually compress each collected binary file (executable, shared library, or python extension) in order to reduce the overall size of the frozen application (the one-dir bundle directory, or the one-file executable). The frozen application's executable itself is not UPX-compressed (regardless of one-dir or one-file mode), as most of its size comprises the embedded archive that already contains individually compressed files.

PyInstaller looks for the UPX in the standard executable path(s) (defined by `PATH` environment variable), or in the path specified via the *--upx-dir* command-line option. If found, it is used automatically. The use of UPX can be completely disabled using the *--noupx* command-line option.

Note: UPX is currently used only on Windows. On other operating systems, the collected binaries are not processed even if UPX is found. The shared libraries (e.g., the Python shared library) built on modern linux distributions seem to break when processed with UPX, resulting in defunct application bundles. On macOS, UPX currently fails to process .dylib shared libraries; furthermore the UPX-compressed files fail the validation check of the `codesign` utility, and therefore cannot be code-signed (which is a requirement on the Apple M1 platform).

Excluding problematic files from UPX processing

Using UPX may end up corrupting a collected shared library. Known examples of such corruption are Windows DLLs with Control Flow Guard (CFG) enabled, as well as Qt5 and Qt6 plugins. In such cases, individual files may be need to be excluded from UPX processing, using the *--upx-exclude* option (or using the `upx_exclude` argument in the *.spec file*).

Changed in version 4.2: PyInstaller detects CFG-enabled DLLs and automatically excludes them from UPX processing.

Changed in version 4.3: PyInstaller automatically excludes Qt5 and Qt6 plugins from UPX processing.

Although PyInstaller attempts to automatically detect and exclude some of the problematic files from UPX processing, there are cases where the UPX excludes need to be specified manually. For example, 32-bit Windows binaries from the `PySide2` package (Qt5 DLLs and python extension modules) have been reported to be corrupted by UPX.

Changed in version 5.0: Unlike earlier releases that compared the provided UPX-exclude names against basenames of the collect binary files (and, due to incomplete case normalization, required provided exclude names to be lowercase on Windows), the UPX-exclude pattern matching now uses OS-default case sensitivity and supports the wildcard (*) operator. It also supports specifying (full or partial) parent path of the file.

The provided UPX exclude patterns are matched against *source* (origin) paths of the collected binary files, and the matching is performed from right to left.

For example, to exclude Qt5 DLLs from the PySide2 package, use `--upx-exclude "Qt*.dll"`, and to exclude the python extensions from the PySide2 package, use `--upx-exclude "PySide2\*.pyd"`.

2.6.5 Splash Screen *(Experimental)*

Note: This feature is incompatible with macOS. In the current design, the splash screen operates in a secondary thread, which is disallowed by the Tcl/Tk (or rather, the underlying GUI toolkit) on macOS.

Some applications may require a splash screen as soon as the application (bootloader) has been started, because especially in onefile mode large applications may have long extraction/startup times, while the bootloader prepares everything, where the user cannot judge whether the application was started successfully or not.

The bootloader is able to display a one-image (i.e. only an image) splash screen, which is displayed before the actual main extraction process starts. The splash screen supports non-transparent and hard-cut-transparent images as background image, so non-rectangular splash screens can also be displayed.

Note: Splash images with transparent regions are not supported on Linux due to Tcl/Tk platform limitations. The `-transparentcolor` and `-transparent` wm attributes used by PyInstaller are not available to Linux.

This splash screen is based on Tcl/Tk, which is the same library used by the Python module tkinter. PyInstaller bundles the dynamic libraries of tcl and tk into the application at compile time. These are loaded into the bootloader at startup of the application after they have been extracted (if the program has been packaged as an onefile archive). Since the file sizes of the necessary dynamic libraries are very small, there is almost no delay between the start of the application and the splash screen. The compressed size of the files necessary for the splash screen is about *1.5 MB*.

As an additional feature, text can optionally be displayed on the splash screen. This can be changed/updated from within Python. This offers the possibility to display the splash screen during longer startup procedures of a Python program (e.g. waiting for a network response or loading large files into memory). You can also start a GUI behind the splash screen, and only after it is completely initialized the splash screen can be closed. Optionally, the font, color and size of the text can be set. However, the font must be installed on the user system, as it is not bundled. If the font is not available, a fallback font is used.

If the splash screen is configured to show text, it will automatically (as onefile archive) display the name of the file that is currently being unpacked, this acts as a progress bar.

2.6.6 The `pyi_splash` Module

The splash screen is controlled from within Python by the *`pyi_splash`* module, which can be imported at runtime. This module **cannot** be installed by a package manager because it is part of PyInstaller and is included as needed. This module must be imported within the Python program. The usage is as follows:

```
import pyi_splash

# Update the text on the splash screen
pyi_splash.update_text("PyInstaller is a great software!")
pyi_splash.update_text("Second time's a charm!")

# Close the splash screen. It does not matter when the call
# to this function is made, the splash screen remains open until
# this function is called or the Python program is terminated.
pyi_splash.close()
```

Of course the import should be in a `try ... except` block, in case the program is used externally as a normal Python script, without a bootloader. For a detailed description see *pyi_splash Module (Detailed)*.

2.6.7 Defining the Extraction Location

When building your application in `onefile` mode (see *Bundling to One File* and *How the One-File Program Works*), you might encounter situations where you want to control the location of the temporary directory where the application unpacks itself. For example:

- your application is supposed to be running for long periods of time, and you need to prevent its files from being deleted by the OS that performs periodic clean-up in standard temporary directories.
- your target POSIX system does not use standard temporary directory location (i.e., `/tmp`) and the standard environment variables for temporary directory are not set in the environment.
- the default temporary directory on the target POSIX system is mounted with `noexec` option, which prevents the frozen application from loading the unpacked shared libraries.

The location of the temporary directory can be overridden dynamically, by setting corresponding environment variable(s) before launching the application, or set statically, using the *--runtime-tmpdir* option during the build process.

Using environment variables

The extraction location can be controlled dynamically, by setting the environment variable(s) that PyInstaller uses to determine the temporary directory. This can, for example, be done in a wrapper shell script that sets the environment variable(s) before running the frozen application's executable.

On POSIX systems, the environment variables used for temporary directory location are `TMPDIR`, `TEMP`, and `TMP`, in that order; if none are defined (or the corresponding directories do not exist or cannot be used), `/tmp`, `/var/tmp`, and `/usr/tmp` are used as hard-coded fall-backs, in the specified order. The directory specified via the environment variable must exist (i.e., the application attempts to create only its own directory under the base temporary directory).

On Windows, the default temporary directory location is determined via GetTempPathW function (which looks at `TMP` and `TEMP` environment variables for initial temporary directory candidates).

Using the `--runtime-tmpdir` option

The location of the temporary directory can be set statically, at compile time, using the *--runtime-tmpdir* option. If this option is used, the bootloader will ignore temporary directory locations defined by the OS, and use the specified path. The path can be either absolute or relative (which makes it relative to the current working directory).

Please use this option only if you know what you are doing.

Note: On POSIX systems, PyInstaller's bootloader does **not** perform shell-style environment variable expansion on the path string given via *--runtime-tmpdir* option. Therefore, using environment variables (e.g., ~ or `$HOME`) in the path will **not** work.

2.6.8 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 3.6 version and a Python 3.7 version; or a supported version that uses Qt4 and a development version that uses Qt5 – we recommend you use venv. With *venv* you can maintain different combinations of Python and installed packages, and switch from one combination to another easily. These are called *virtual environments* or *venvs* in short.

- Use *venv* to create as many different development environments as you need, each with its unique combination of Python and installed packages.
- Install PyInstaller in each virtual environment.
- Use PyInstaller to build your application in each virtual environment.

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

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

Under Windows, the pip-Win package makes it especially easy to set up different environments and switch between them. Under GNU/Linux and macOS, you switch environments at the command line.

See **PEP 405** and the official Python Tutorial on Virtual Environments and Packages 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 macOS, 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.

A File Sync & Share system like NextCloud is useful with virtual machines. Install the synchronization client in each virtual machine, all linked to your synchronization account. Keep a single copy of your script(s) in a synchronized folder. Then on any virtual machine you can run PyInstaller thus:

```
cd ~/NextCloud/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 synchronized 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 macOS, 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.

2.6.9 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:

> `pyi-grab_version` *executable_with_version_resource*

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

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

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

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

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

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

> `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.

2.6.10 Building macOS App Bundles

Under macOS, 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 a macOS app bundle named `myscript.app`.

Note: Generating app bundles with onefile executables (i.e., using the combination of *--onefile* and *--windowed* options), while possible, is not recommended. Such app bundles are inefficient, because they require unpacking on each run (and the unpacked content might be scanned by the OS each time). Furthermore, onefile executables will not work when signed/notarized with sandbox enabled (which is a requirement for distribution of apps through Mac App Store).

As you are likely aware, an app bundle is a special type of folder. The one built by PyInstaller always contains a folder named `Contents`, which contains:

- A file named `Info.plist` that describes the app.
- A folder named `MacOS` that contains the program executable.
- A folder named `Frameworks` that contains the collected binaries (shared libraries, python extensions) and nested .framework bundles. It also contains symbolic links to data files and directories from the `Resources` directory.
- A folder named `Resources` that contains the icon file and all collected data files. It also contains symbolic links to binaries and directories from the `Resources` directory.

Note: The contents of the `Frameworks` and `Resources` directories are cross-linked between the two directories in an effort to maintain an illusion of a single content directory (which is required by some packages), while also trying to satisfy the Apple's file placement requirements for codesigning.

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 macOS Bundle* below.

2.6.11 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.

Note that PyInstaller does bundle other shared libraries that are discovered via dependency analysis, such as libstdc++.so.6, libfontconfig.so.1, libfreetype.so.6. These libraries may be required on systems where older (and thus incompatible) versions of these libraries are available. On the other hand, the bundled libraries may cause issues when trying to load a system-provided shared library that is linked against a newer version of the system-provided library.

For example, system-installed mesa DRI drivers (e.g., radeonsi_dri.so) depend on the system-provided version of libstdc++.so.6. If the frozen application bundles an older version of libstdc++.so.6 (as collected from the build system), this will likely cause missing symbol errors and prevent the DRI drivers from loading. In this case, the bundled libstdc++.so.6 should be removed. However, this may not work on a different distribution that provides libstdc++.so.6 older than the one from the build system; in that case, the bundled version should be kept, because the system-provided version may lack the symbols required by other collected binaries that depend on libstdc++.so.6.

Windows

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.

macOS

Making macOS apps Forward-Compatible

On macOS, system components from one version of the OS are usually compatible with later versions, but they may not work with earlier versions. While PyInstaller does not collect system components of the OS, the collected 3rd party binaries (e.g., python extension modules) are built against specific version of the OS libraries, and may or may not support older OS versions.

As such, the only way to ensure that your frozen application supports an older version of the OS is to freeze it on the oldest version of the OS that you wish to support. This applies especially when building with Homebrew python, as its binaries usually explicitly target the running OS.

For example, to ensure compatibility with "Mojave" (10.14) and later versions, you should set up a full environment (i.e., install python, PyInstaller, your application's code, and all its dependencies) in a copy of macOS 10.14, using a virtual machine if necessary. Then use PyInstaller to freeze your application in that environment; the generated frozen application should be compatible with that and later versions of macOS.

Building 32-bit Apps in macOS

Note: This section is largely obsolete, as support for 32-bit application was removed in macOS 10.15 Catalina (for 64-bit multi-arch support on modern versions of macOS, see *here*). However, PyInstaller still supports building 32-bit bootloader, and 32-bit/64-bit Python installers are still available from python.org for (some) versions of Python 3.7 which PyInstaller dropped support for in v6.0.

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

To verify that the installed python version supports execution in either 64- or 32-bit mode, use the `file` command on the Python executable:

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

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

```
$ /usr/local/bin/python3
Python 3.7.6 (v3.7.6:43364a7ae0, Dec 18 2019, 14:12:53)
[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 3.7.6 (v3.7.6:43364a7ae0, Dec 18 2019, 14:12:53)
[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
```

Note: PyInstaller does not provide pre-built 32-bit bootloaders for macOS anymore. In order to use PyInstaller with 32-bit python, you need to *build the bootloader* yourself, using an XCode version that still supports compiling 32-bit. Depending on the compiler/toolchain, you may also need to explicitly pass `--target-arch=32bit` to the `waf` command.

Getting the Opened Document Names

When user double-clicks a document of a type that is registered with your application, or when a user drags a document and drops it on your application's icon, macOS launches your application and provides the name(s) of the opened document(s) in the form of an OpenDocument AppleEvent.

These events are typically handled via installed event handlers in your application (e.g., using `Carbon` API via `ctypes`, or using facilities provided by UI toolkits, such as `tkinter` or `PyQt5`).

Alternatively, PyInstaller also supports conversion of open document/URL events into arguments that are appended to `sys.argv`. This applies only to events received during application launch, i.e., before your frozen code is started. To handle events that are dispatched while your application is already running, you need to set up corresponding event handlers.

For details, see *this section*.

AIX

Depending on whether Python was build as a 32-bit or a 64-bit executable you may need to set or unset the environment variable `OBJECT_MODE`. To determine the size the following command can be used:

```
$ python -c "import sys; print(sys.maxsize <= 2**32)"
True
```

When the answer is `True` (as above) Python was build as a 32-bit executable.

When working with a 32-bit Python executable proceed as follows:

```
$ unset OBJECT_MODE
$ pyinstaller <your arguments>
```

When working with a 64-bit Python executable proceed as follows:

```
$ export OBJECT_MODE=64
$ pyinstaller <your arguments>
```

2.7 Common Issues and Pitfalls

This section attempts to document common issues and pitfalls that users need to be aware of when trying to freeze their applications with PyInstaller, as certain features require special care and considerations that might not be obvious when developing and running unfrozen python programs.

2.7.1 Requirements Imposed by Symbolic Links in Frozen Application

Starting with PyInstaller 6.0, the frozen application bundles generated by PyInstaller on non-Windows systems make extensive use of symbolic links. Therefore, creation and distribution of PyInstaller-frozen applications requires special considerations.

Failing to preserve symbolic links will turn them into full file copies; the duplicated files will balloon the size of your frozen application, and may also lead to run-time issues.

Note: In PyInstaller versions prior to 6.0, symbolic links were used only in generated *macOS .app bundles*. From 6.0 on, they are also used in "regular" POSIX builds on all POSIX systems (macOS, Linux, FreeBSD, etc.), both in `onefile` and `onedir` mode.

In `onefile` builds, the use of symbolic links imposes run-time requirements for the temporary directory into which the program unpacks itself before running - **the temporary directory must be located on filesystem that supports symbolic links**. Otherwise, the program will fail to unpack itself, as it will encounter an error when trying to (re)create a symbolic link.

The `onedir` builds can only be generated on a filesystem that supports symbolic links. Similarly, they can only be moved or copied to a filesystem that supports symbolic links. If you plan to distribute your `onedir` application as an archive, ensure that archive format supports preservation of symbolic links.

Note: When copying the generated `onedir` application bundle, ensure that you use copy command with options that preserve symbolic links. For example, on Linux, both `cp -fr <source> <dest>` and `cp -fR <source> <dest>` preserve symbolic links. On macOS, on the other hand, `cp -fr <source> <dest>` **does not** preserve symbolic links, while `cp -fR <source> <dest>` does.

Note: Creation of a `zip` archive by default **does not** preserve symbolic links; preservation needs to be explicitly enabled via `--symlinks` / `-y` command-line switch to the `zip` command.

2.7.2 Launching External Programs from the Frozen Application

In a PyInstaller-frozen application, the run-time environment of the application's process is often modified to ensure that when it comes to loading of the shared libraries, the bundled copies of shared libraries are preferred over the copies that might be available on the target system. The exact way of modifying the library search order (environment variables versus low-level API) depends on the operating system, but in general, changes made to the frozen application's run-time environment are also inherited by subprocesses launched by the frozen application. This ensures that the application itself (for example, the binary python extensions it loads) as well as bundled helper programs that the application might run as a subprocess (for example, `gspawn` when using `GLib/Gio` via `gi.repository` on Windows, `QtWebEngineHelper` from `PyQt` and `PySide` packages, and so on) use the shared libraries they they were originally built against and thus have compatible ABI. This makes frozen applications portable, more or less self-contained, and isolated from the target environment.

The above paradigm is inherently at odds with code that is trying to launch an **external program**, i.e., a program that is available on the target system (and launched, for example, via `subprocess.run()`). System-installed external programs are built against shared libraries provided by the system, which might be of different and incompatible versions compared to the ones bundled with the frozen application. And because in the run-time environment of the PyInstaller-frozen application (which is inherited by the launched subprocesses) the library search path is modified to prefer the bundled shared libraries, trying to launch an external program might fail due to shared library conflicts.

Therefore, if your code (or the 3rd party code you are using) is trying to launch an external program, you need to ensure that the changes to the library search paths, made for the frozen application's main process, are reset or reverted. The specifics of such **run-time environment sanitization** are OS-dependent, and are outlined in the following sub-sections.

Note: On some operating systems, the library search path is modified only via environment variables; in such cases, if you are launching the subprocess in your code (e.g., via `subprocess.run()`), you can pass the sanitized environment to the subprocess via the `env` argument. This way, only the environment of the sub-process is modified, while the environment of the frozen application itself (i.e., its search paths) are left unchanged.

If this is not possible, however, you might need to temporarily sanitize the environment of the main application, launch the external program (so it inherits the sanitized environment), and then restore the main application's environment back to the original (PyInstaller-adjusted) version.

If you are launching the external program *after* all modules have been imported and their dependencies have been loaded, and if your frozen application does not include any helper programs that might be launched after your external program, you can simply sanitize main application's run-time environment, without having to worry about restoring it after your external program is launched.

Linux and Unix-like OSes

On POSIX systems (with exception of macOS - see its dedicated sub-section), the library search path is modified via the `LD_LIBRARY_PATH` environment variable (`LIBPATH` on AIX).

During the frozen application's startup, the PyInstaller's bootloader checks whether the `LD_LIBRARY_PATH` environment variable is already set, and, if necessary, creates a copy of its contents into `LD_LIBRARY_PATH_ORIG` environment variable. Then, it modifies `LD_LIBRARY_PATH` by prepending the application's top level directory (i.e., the path that is also available in `sys._MEIPASS`).

Therefore, prior to launching an external program, the `LD_LIBRARY_PATH` should be either cleared (to use the system default) or reset to the value stored in `LD_LIBRARY_PATH_ORIG` (if available). See *LD_LIBRARY_PATH / LIBPATH considerations* for details and an example.

Windows

On Windows, the PyInstaller's bootloader sets the library search path to the top-level application directory (i.e., the path that is also available in `sys._MEIPASS`) using the SetDllDirectoryW Win32 API function.

As noted in the API documentation, calling this function also affects the children processes started from the frozen application. To undo the effect of this call and restore standard search paths, `SetDllDirectory` function should be called with `NULL` argument. As discussed in #3795, the most practical way to achieve this from python code is to use `ctypes`, for example:

```
import sys
if sys.platform == "win32":
    import ctypes
    ctypes.windll.kernel32.SetDllDirectoryW(None)
```

PyInstaller's bootloader does not modify the `PATH` environment variable. However, the `PATH` environment variable may be modified by run-time hooks for specific packages, in order to facilitate discovery of dynamic dependencies that are loaded at run-time.

Therefore, it may also be necessary to sanitize the `PATH` environment variable, and (temporarily) remove any paths anchored in top-level application directory (`sys._MEIPASS`) prior to launching the external program.

macOS

On macOS, PyInstaller rewrites the library paths in collected binaries to refer to copies (or symbolic links) in the top-level application directory, relative to the binary's location. Therefore, PyInstaller's bootloader does not need to modify the `DYLD_LIBRARY_PATH` environment variable.

However, the `DYLD_LIBRARY_PATH` environment variable may be modified by run-time hooks for specific packages, in order to facilitate discovery of dynamic dependencies that are loaded at run-time.

Therefore, it may also be necessary to sanitize the `DYLD_LIBRARY_PATH` environment variable, and (temporarily) remove any paths anchored in top-level application directory (`sys._MEIPASS`) prior to launching the external program.

Note: If you are building a macOS .app bundle, you should be aware that when launched from Finder, the app process runs in an environment with reduced set of environment variables. Most notably, the `PATH` environment variable is set to only `/usr/bin:/bin:/usr/sbin:/sbin`. Therefore, programs installed in locations that are typically in `PATH` when running a Terminal session (e.g., `/usr/local/bin`, `/opt/homebrew/bin`) will not be visible to the app, unless referenced by their full path.

2.7.3 Multi-processing

Currently, the only supported multi-processing framework is the `multiprocessing` module from the Python standard library, and even that requires you to make a `multiprocessing.freeze_support()` call before using any `multiprocessing` functionality.

A typical symptom of failing to call `multiprocessing.freeze_support()` before your code (or 3rd party code you are using) attempts to make use of `multiprocessing` functionality is an endless spawn loop of your application process.

Note: `multiprocessing` supports different start modes: `spawn`, `fork`, and `forkserver`. Of these, `fork` is the only one that might work in the frozen application without calling `multiprocessing.freeze_support()`. The default start method on Windows and macOS is `spawn`, while `fork` is default on other POSIX systems (however, Python 3.14 plans to change that).

Why is calling *multiprocessing.freeze_support()* required?

As implied by its name, the `multiprocessing` module spawns several processes; typically, these are worker processes running your tasks. On POSIX systems, `spawn` and `forkserver` start methods also spawn a dedicated resource tracker process that tracks and handles clean-up of unlinked shared resources (e.g., shared memory segments, semaphores).

The sub-processes started by `multiprocessing` are spawned using `sys.executable` - when running an unfrozen python script, this corresponds to your python interpreter executable (e.g., `python.exe`). The command-line arguments instruct the interpreter to run a corresponding function from the `multiprocessing` module. For example, the spawned worker process on Windows looks as follows:

```
python.exe -c "from multiprocessing.spawn import spawn_main; spawn_main(parent_pid=6872,␣
↪pipe_handle=520)" --multiprocessing-fork
```

Similarly, when using the `spawn` start method on a POSIX system, the resource tracker process is started with the following arguments:

```
python -c from multiprocessing.resource_tracker import main;main(5)
```

while the worker process is started with the following arguments:

```
python -c "from multiprocessing.spawn import spawn_main; spawn_main(tracker_fd=6, pipe_
↪handle=8)" --multiprocessing-fork
```

In the frozen application, `sys.executable` points to your application executable. So when the `multiprocessing` module in your main process attempts to spawn a subprocess (a worker or the resource tracker), it runs another instance of your program, with the following arguments for resource tracker:

```
my_program -B -S -I -c "from multiprocessing.resource_tracker import main;main(5)"
```

and for the worker process:

```
my_program --multiprocessing-fork tracker_fd=6 pipe_handle=8
```

On Windows, the worker process looks similar:

```
my_program.exe --multiprocessing-fork parent_pid=8752 pipe_handle=1552
```

If no special handling is in place in the program code, the above invocations end up executing your program code, which leads to one of the two outcomes:

- this second program instance again reaches the point where `multiprocessing` module attempts to spawn a subprocess, leading to an endless recursive spawn loop that eventually crashes your system.
- if you have command-line parsing implemented in your program code, the command-line parser raises an error about unrecognized parameters. Which may lead to periodic attempts at spawning the resource tracker process.

Enter `multiprocessing.freeze_support()` - PyInstaller provides a custom override of this function, which **is required to be called on all platforms** (in contrast to original standard library implementation, which, as suggested by its documentation, caters only to Windows). Our implementation inspects the arguments (`sys.argv`) passed to the process, and if they match the arguments used by `multiprocessing` for a worker process or resource tracker, it diverts the program flow accordingly (i.e., executes the corresponding `multiprocessing` code and exits after finished execution).

This ensures that `multiprocessing` sub-processes, while re-using the application executable, execute their intended `multiprocessing` functionality instead of executing your main program code.

When to call *multiprocessing.freeze_support()*?

The rule of thumb is, `multiprocessing.freeze_support()` should be called before trying to use any of `multiprocessing` functionality (such as spawning a process or opening process pool, or allocating a shared resource, for example a semaphore).

Therefore, as documented in original implementation of `multiprocessing.freeze_support()`, a typical call looks like this:

```
from multiprocessing import Process, freeze_support

def f():
    print('hello world!')

if __name__ == '__main__':
    freeze_support()
    Process(target=f).start()
```

However, there are scenarios where you might need to make the call even sooner, before (at least some of) the imports at the top of your script. This might be necessary if your script imports a module that does one of the following during its initialization (i.e., when it is imported):

- makes use of `multiprocessing` functionality.
- parses command-line arguments for your program.
- imports and initializes a GUI framework. While this might not result in an error, it should be avoided in the worker processes by diverting the program flow before it happens.

Similarly, if both of the following conditions are true:

- your script imports several heavy-weight modules that are needed by the main program but not by the worker process
- your script does not directly use `multiprocessing` functionality itself, but rather imports a 3rd party module and calls a function from it that uses `multiprocessing`,

then it might be worth placing the `multiprocessing.freeze_support()` before the imports, to avoid unnecessarily slowing the worker processes:

```
# Divert the program flow in worker sub-process as soon as possible,
# before importing heavy-weight modules.
if __name__ == '__main__':
    import multiprocessing
    multiprocessing.freeze_support()

# Import several heavy-weight modules
import numpy as np
import cv2
# ...
import some_module

if __name__ == '__main__':
    # Call some 3rd party function that internally uses multiprocessing
    some_module.some_function_that_uses_multiprocessing()
```

Note: If `multiprocessing` is used only in an external module that is imported and used by your script, then the `multiprocessing` worker sub-process needs to load and initialize only that module; therefore, diverting the program flow using `multiprocessing.freeze_support()` before performing heavy-weight imports avoids unnecessarily slowing down the worker process.

On the other hand, if your main script (also) uses `multiprocessing` functionality, then the corresponding worker sub-process also need to execute the remainder of your script, including the imports; which limits the performance benefits of an early `multiprocessing.freeze_support()` call.

What about other multi-processing frameworks?

The Python ecosystem provides several alternatives to the `multiprocessing` from the Python standard library - **none of them are supported by PyInstaller**.

The PyInstaller-frozen application does not have access to python interpreter executable (`python` or `python.exe`) and its environment, and must therefore use its embedded python interpreter. Therefore, any other alternative python-based multi-processing solution would also need to spawn its worker subprocesses using the program executable (`sys.executable`).

Even if the alternative multi-processing framework uses `sys.executable` to spawn its subprocesses, your program code would need to be made aware of such attempts, and handle them accordingly. In other words, you would need to implement inspection of program arguments (`sys.argv`), detect attempts at spawning worker subprocesses based on the arguments, and divert the program flow into corresponding framework's function instead of letting it reach your main program code.

2.7.4 `sys.stdin`, `sys.stdout`, and `sys.stderr` in `noconsole`/`windowed` Applications (Windows only)

On Windows, the *`--noconsole`* allows you to build a frozen application using the "windowed" bootloader variant, which was built with `/SUBSYSTEM:WINDOWED` option (as opposed to `/SUBSYSTEM:CONSOLE`; see here for details), and thus has no console attached. This is similar to the *windowed python interpreter executable*, `pythonw.exe`, which can be used to run python scripts that do not require a console, nor want to open a console window when launched.

A direct consequence of building your frozen application in the windowed/no-console mode is that standard input/output file objects, `sys.stdin`, `sys.stdout`, and `sys.stderr` are unavailable, and are set to `None`. The same would happen if you ran your unfrozen code using the `pythonw.exe` interpreter, as documented under `sys.__stderr__` in Python standard library documentation.

Therefore, if your code (or the 3rd party code you are using) naively attempts to access attributes of `sys.stdout` and `sys.stderr` objects without first ensuring that the objects are available, the frozen application will raise an `AttributeError`; for example, trying to access `sys.stderr.flush` will result in `'NoneType' object has no attribute 'flush'`.

The best practice would be to fix the offending code so that it checks for availability of the standard I/O file objects before trying to use them; this will ensure compatibility with both `pythonw.exe` interpreter and with PyInstaller's `noconsole` mode. However, if fixing the problem is not an option (for example, the problem originates from a 3rd party module and is beyond your control), you can work around it by setting dummy file handles at the very start of your program:

```
import sys
import os

if sys.stdout is None:
    sys.stdout = open(os.devnull, "w")
if sys.stderr is None:
    sys.stderr = open(os.devnull, "w")

# the rest of your imports

# and the rest of your program
```

Note: If you plan to build your frozen application in windowed/no-console mode, we recommend that you first try running your unfrozen script using the `pythonw.exe` interpreter to ensure that it works correctly when console is

unavailable.

2.8 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?":

```
import sys
if getattr(sys, 'frozen', False) 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 the `_internal` folder within the bundle. 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.

2.8.1 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):

```
from os import path
path_to_dat = path.abspath(path.join(path.dirname(__file__), 'file.dat'))
```

In the main script (the `__main__` module) itself, the `__file__` variable contains path to the script file. In Python 3.8 and earlier, this path is either absolute or relative (depending on how the script was passed to the `python` interpreter), while in Python 3.9 and later, it is always an absolute path. In the bundled script, the PyInstaller bootloader always sets the `__file__` variable inside the `__main__` module to the absolute path inside the bundle directory, as if the byte-compiled entry-point script existed there.

For example, if your entry-point script is called `program.py`, then the `__file__` attribute inside the bundled script will point to `sys._MEIPASS + 'program.py'`. Therefore, locating a data file relative to the main script can be either done directly using `sys._MEIPASS` or via the parent path of the `__file__` inside the main script.

The following example will get the path to a file `other-file.dat` located next to the main script if not bundled and inside the bundle folder if it is bundled:

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

Or, if you'd rather use pathlib:

```
from pathlib import Path
path_to_dat = Path(__file__).resolve().with_name("other-file.dat")
```

Changed in version 4.3: Formerly, the `__file__` attribute of the entry-point script (the `__main__` module) was set to only its basename rather than its full (absolute or relative) path within the bundle directory. Therefore, PyInstaller documentation used to suggest `sys._MEIPASS` as means for locating resources relative to the bundled entry-point script. Now, `__file__` is always set to the absolute full path, and is the preferred way of locating such resources.

Placing data files at expected locations inside the bundle

To place the data-files where your code expects them to be (i.e., relative to the main script or bundle directory), you can use the **dest** parameter of the *--add-data="source:dest"* command-line switches. Assuming you normally use the following code in a file named `my_script.py` to locate a file `file.dat` in the same folder:

```
from os import path
path_to_dat = path.abspath(path.join(path.dirname(__file__), 'file.dat'))
```

Or the pathlib equivalent:

```
from pathlib import Path
path_to_dat = Path(__file__).resolve().with_name("file.dat")
```

And `my_script.py` is **not** part of a package (not in a folder containing an `__init_.py`), then `__file__` will be `[app root]/my_script.pyc` meaning that if you put `file.dat` in the root of your package, using:

```
PyInstaller --add-data="/path/to/file.dat:."
```

It will be found correctly at runtime without changing `my_script.py`.

If `__file__` is checked from inside a package or library (say `my_library.data`) then `__file__` will be `[app root]/my_library/data.pyc` and *--add-data* should mirror that:

```
PyInstaller --add-data="/path/to/my_library/file.dat:./my_library"
```

However, in this case it is much easier to switch to *the spec file* and use the *PyInstaller.utils.hooks.collect_data_files()* helper function:

```
from PyInstaller.utils.hooks import collect_data_files

a = Analysis(...,
             datas=collect_data_files("my_library"),
             ...)
```

2.8.2 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:

```
#!/usr/bin/env 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() )
```

2.8.3 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*.

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.

```
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')
```

(continues on next page)

(continued from previous page)

```
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
```

See also: *Launching External Programs from the Frozen Application*

2.9 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*
- *--clean*
- *--log-level*

2.9.1 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 a shortened example of a spec file for a minimal, one-folder app:

```
a = Analysis(['minimal.py'],
         pathex=['/Developer/PItests/minimal'],
         binaries=None,
         datas=None,
         hiddenimports=[],
         hookspath=None,
         runtime_hooks=None,
         excludes=None)
pyz = PYZ(a.pure)
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;
  - `pathex`: a list of paths to search for imports (like using `PYTHONPATH`), including paths given by the *--paths* option.
  - `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`.

2.9.2 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:

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

And the command line equivalent:

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
        ...
        )
```

You can also include the entire contents of a folder:

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

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

Using Data Files from a Module

If the data files you are adding are contained within a Python module, you can retrieve them using `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:

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

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

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

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

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

Adding Binary Files

Note: *Binary* files refers to DLLs, dynamic libraries, shared object-files, and such, which PyInstaller is going to search for further *binary* dependencies. Files like images and PDFs should go into the `datas`.

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.

Adding binary files works in a similar way as adding data files. As described in *Adding Binary Files*, each tuple should have two values:

- 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.

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

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

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

Or via the command line:

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

If you wish to store `libiodbc.2.dylib` on a specific folder inside the bundle, for example `vendor`, then you could specify it, using the second element of the tuple:

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

As with data files, if you have multiple binary files to add, to improve readability, 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 Table of Contents (TOC) lists and the Tree Class* below.

2.9.3 Specifying Python Interpreter Options

PyInstaller-frozen application runs the application code in isolated, embedded Python interpreter. Therefore, **the typical means of passing options to Python interpreter do not apply**, including:

- environment variables (such as *PYTHONUTF8* and *PYTHONHASHSEED*) - because the frozen application is supposed to be isolated from python environment that might be present on the target system
- command-line arguments (such as *-v* and *-O*) - because command-line arguments are reserved for application.

Instead, PyInstaller offers an option to specify permanent run-time options for the application's Python interpreter via its own `OPTIONS` mechanism. To pass run-time options, create a list of three-element tuples: *('option string', None, 'OPTION')*, and pass it as an additional argument to *EXE* before the keyword arguments. The first element of the option tuple is the option string (see below for valid options), the second is always *None*, and the third is always *'OPTION'*.

An example spec file, modified to specify two run-time options:

```
options = [
    ('v', None, 'OPTION'),
    ('W ignore', None, 'OPTION'),
]

a = Analysis(
    ...
)
...
exe = EXE(
    pyz,
    a.scripts,
    options,  # <-- the options list, passed to EXE
    exclude_binaries=...
    ...
)
```

The following options are supported by this mechanism:

- `'v'` or `'verbose'`: increment the value of `sys.flags.verbose`, which causes messages to be written to stdout each time a module is initialized. This option is equivalent to Python's `-v` command-line option. It is automatically enabled when *verbose imports* are enabled via PyInstaller's own ``–debug imports`` option.
- `'u'` or `'unbuffered'`: enable unbuffered stdout and stderr. Equivalent to Python's `-u` command-line option.
- `'O'` or `'optimize'`: increment the value of `sys.flags.optimize`. Equivalent to Python's `-O` command-line option.

Note: The optimization level reflected by `sys.flags.optimize` affects only bytecode that python interpreter would end up compiling at the run time. In a PyInstaller-frozen application, however, most of python modules are available as pre-compiled bytecode, hence the run-time bytecode optimization level does not affect them at all.

For details on how to enforce the bytecode optimization level for collected modules, see *Bytecode Optimization Level*.

- `'W <arg>'`: a pass-through for Python's W-options that control warning messages.
- `'X <arg>'`: a pass-through for Python's X-options. The `utf8` and `dev` X-options, which control UTF-8 mode and developer mode, are explicitly parsed by PyInstaller's bootloader and used during interpreter pre-initialization; the rest of X-options are just passed on to the interpreter configuration.

- 'hash_seed=<value>': an option to set Python's hash seed within the frozen application to a fixed value. Equivalent to PYTHONHASHSEED environment variable. At the time of writing, this does not exist as an X-option, so it is implemented as a custom option.

Further examples to illustrate the syntax:

```
options = [
    # Warning control
    ('W ignore', None, 'OPTION'),  # disable all warnings
    ('W ignore::DeprecationWarning', None, 'OPTION')  # disable deprecation warnings

    # UTF-8 mode; unless explicitly enabled/disabled, it is auto enabled based on locale
    ('X utf8', None, 'OPTION),  # force UTF-8 mode on
    ('X utf8=1', None, 'OPTION),  # force UTF-8 mode on
    ('X utf8=0', None, 'OPTION),  # force UTF-8 mode off

    # Developer mode; disabled by default
    ('X dev', None, 'OPTION),  # enable dev mode
    ('X dev=1', None, 'OPTION),  # enable dev mode

    # Hash seed
    ('hash_seed=0', None, 'OPTION'),  # disable hash randomization; sys.flags.hash_
↪randomization=0
    ('hash_seed=123', None, 'OPTION'),  # hash randomization with fixed seed value
]
```

2.9.4 Spec File Options for a macOS Bundle

When you build a windowed macOS app (that is, running under macOS, you specify the *--windowed* option), the spec file contains an additional statement to create the macOS application bundle, or app folder:

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

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

An Info.plist file is an important part of a macOS app bundle. (See the Apple bundle overview for a discussion of the contents of Info.plist.)

PyInstaller creates a minimal Info.plist. The version option can be used to set the application version using the CFBundleShortVersionString Core Foundation Key.

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:

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

(continues on next page)

(continued from previous page)

```
          bundle_identifier=None,
          version='0.0.1',
          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 macOS 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 macOS what filetypes your application supports (see Apple document types).

2.9.5 POSIX Specific Options

By default all required system libraries are bundled. To exclude all or most non-Python shared system libraries from the bundle, you can add a call to the function `exclude_system_libraries` from the Analysis class. System libraries are defined as files that come from under `/lib*` or `/usr/lib*` as is the case on POSIX and related operating systems. The function accepts an optional parameter that is a list of file wildcards exceptions, to not exclude library files that match those wildcards in the bundle. For example to exclude all non-Python system libraries except "libexpat" and anything containing "krb" use this:

```
a = Analysis(...)

a.exclude_system_libraries(list_of_exceptions=['libexpat*', '*krb*'])
```

2.9.6 The `Splash` Target

For a splash screen to be displayed by the bootloader, the `Splash` target must be called at build time. This class can be added when the spec file is created with the command-line option *`--splash IMAGE_FILE`*. By default, the option to display the optional text is disabled (`text_pos=None`). For more information about the splash screen, see *Splash Screen (Experimental)* section. The `Splash` Target looks like this:

```
a = Analysis(...)

splash = Splash('image.png',
                binaries=a.binaries,
                datas=a.datas,
                text_pos=(10, 50),
                text_size=12,
                text_color='black')
```

Splash bundles the required resources for the splash screen into a file, which will be included in the CArchive.

A `Splash` has two outputs, one is itself and one is stored in `splash.binaries`. Both need to be passed on to other build targets in order to enable the splash screen. To use the splash screen in a **onefile** application, please follow this example:

```
a = Analysis(...)

splash = Splash(...)

# onefile
exe = EXE(pyz,
          a.scripts,
          splash,                  # <-- both, splash target
          splash.binaries,         # <-- and splash binaries
          ...)
```

In order to use the splash screen in a **onedir** application, only a small change needs to be made. The `splash.binaries` attribute has to be moved into the `COLLECT` target, since the splash binaries do not need to be included into the executable:

```
a = Analysis(...)

splash = Splash(...)

# onedir
exe = EXE(pyz,
          splash,                  # <-- splash target
          a.scripts,
          ...)
coll = COLLECT(exe,
               splash.binaries,    # <-- splash binaries
               ...)
```

On Windows/macOS images with per-pixel transparency are supported. This allows non-rectangular splash screen images. On Windows the transparent borders of the image are hard-cuted, meaning that fading transparent values are not supported. There is no common implementation for non-rectangular windows on Linux, so images with per- pixel transparency is not supported.

The splash target can be configured in various ways. The constructor of the `Splash` target is as follows:

`Splash.__init__`(*image_file*, *binaries*, *datas*, ***kwargs*)

Parameters

- **`image_file`** (*str*) – A path-like object to the image to be used. Only the PNG file format is supported.

  Note: If a different file format is supplied and PIL (Pillow) is installed, the file will be converted automatically.

  Note: *Windows*: The color `'magenta'` / `'#ff00ff'` must not be used in the image or text, as it is used by splash screen to indicate transparent areas. Use a similar color (e.g., `'#ff00fe'`) instead.

Note: If PIL (Pillow) is installed and the image is bigger than max_img_size, the image will be resized to fit into the specified area.

- **binaries** (*list*) – The TOC list of binaries the Analysis build target found. This TOC includes all extension modules and their binary dependencies. This is required to determine whether the user's program uses *tkinter*.
- **datas** (*list*) – The TOC list of data the Analysis build target found. This TOC includes all data-file dependencies of the modules. This is required to check if all splash screen requirements can be bundled.

Keyword Arguments

- **text_pos** – An optional two-integer tuple that represents the origin of the text on the splash screen image. The origin of the text is its lower left corner. A unit in the respective coordinate system is a pixel of the image, its origin lies in the top left corner of the image. This parameter also acts like a switch for the text feature. If omitted, no text will be displayed on the splash screen. This text will be used to show textual progress in onefile mode.
- **text_size** – The desired size of the font. If the size argument is a positive number, it is interpreted as a size in points. If size is a negative number, its absolute value is interpreted as a size in pixels. Default: `12`
- **text_font** – An optional name of a font for the text. This font must be installed on the user system, otherwise the system default font is used. If this parameter is omitted, the default font is also used.
- **text_color** – An optional color for the text. HTML color codes (`'#40e0d0'`) and color names (`'turquoise'`) are supported. Default: `'black'` (Windows: the color `'magenta'` / `'#ff00ff'` is used to indicate transparency, and should not be used)
- **text_default** – The default text which will be displayed before the extraction starts. Default: `"Initializing"`
- **full_tk** – By default Splash bundles only the necessary files for the splash screen (some tk components). This options enables adding full tk and making it a requirement, meaning all tk files will be unpacked before the splash screen can be started. This is useful during development of the splash screen script. Default: `False`
- **minify_script** – The splash screen is created by executing an Tcl/Tk script. This option enables minimizing the script, meaning removing all non essential parts from the script. Default: `True`
- **rundir** – The folder name in which tcl/tk will be extracted at runtime. There should be no matching folder in your application to avoid conflicts. Default: `'__splash'`
- **name** – An optional alternative filename for the .res file. If not specified, a name is generated.
- **script_name** – An optional alternative filename for the Tcl script, that will be generated. If not specified, a name is generated.
- **max_img_size** – Maximum size of the splash screen image as a tuple. If the supplied image exceeds this limit, it will be resized to fit the maximum width (to keep the original aspect ratio). This option can be disabled by setting it to None. Default: `(760, 480)`
- **always_on_top** – Force the splashscreen to be always on top of other windows. If disabled, other windows (e.g., from other applications) can cover the splash screen by user bringing them to front. This might be useful for frozen applications with long startup times. Default: `True`

2.9.7 Multipackage Bundles

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

You can use the multipackage feature to bundle a set of executable apps so that they share single copies of libraries. You can do this with either one-file or one-folder apps.

Multipackaging with One-Folder Apps

For combining multiple one-folder applications, use a shared COLLECT statement. This will collect the external resources for all of the one-folder apps into one directory.

Multipackaging with One-File 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. 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`. 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.` Modify the EXE statements to pass in `Analysis.dependencies`, in addition to all other arguments that are passed in the original EXE statements. For example:

```
foo_pyz = PYZ(foo_a.pure)
foo_exe = EXE(foo_pyz, foo_a.dependencies, foo_a.scripts, ... etc.

bar_pyz = PYZ(bar_a.pure)
bar_exe = EXE(bar_pyz, bar_a.dependencies, bar_a.scripts, ... etc.
```

Save the combined spec file as `foobarzap.spec` and then build it:

```
pyinstaller foobarzap.spec
```

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

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 Table of Contents (TOC) lists and the Tree Class* described below.

2.9.8 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`, `Tree` and `Splash`, which are discussed in the preceding sections.

Other globals contain information about the build environment:

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

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

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

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

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

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

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

2.9.9 Adding parameters to spec files

Sometimes, you may wish to have different build modes (e.g. a *debug* build and a *production* build) from the same spec file. Any command line arguments to `pyinstaller` given after a `--` separator will not be parsed by PyInstaller and will instead be forwarded to the spec file where you can implement your own argument parsing and handle the options accordingly. For example, the following spec file will create a onedir application with console enabled if invoked via `pyinstaller example.spec -- --debug` or a onefile console-less application if invoked with just `pyinstaller example.spec`. If you use an `argparse` based parser rather than rolling your own using `sys.argv` then `pyinstaller example.spec -- --help` will display your spec options.

```
# example.spec

import argparse

parser = argparse.ArgumentParser()
parser.add_argument("--debug", action="store_true")
options = parser.parse_args()

a = Analysis(
    ['example.py'],
)
pyz = PYZ(a.pure)

if options.debug:
    exe = EXE(
        pyz,
        a.scripts,
        exclude_binaries=True,
        name='example',
    )
    coll = COLLECT(
```

(continues on next page)

(continued from previous page)

```
        exe,
        a.binaries,
        a.datas,
        name='example_debug',
    )
else:
    exe = EXE(
        pyz,
        a.scripts,
        a.binaries,
        a.datas,
        name='example',
        console=False,
    )
```

2.10 Notes about specific Features

2.10.1 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:

```
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:

```
CDLL("library.so")
WinDLL("library.so")
ctypes.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.

2.10.2 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, 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.

2.10.3 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*.

2.10.4 Bytecode Optimization Level

In unfrozen Python, the PYTHONOPTIMIZE environment variable and the -O command-line option control the optimization level, which is reflected in the value of the optimize flag in sys.flags. The optimization level determines how python byte-compiles pure-python modules when it loads the for the first time (or which version of byte-compiled modules is loaded from __pycache__, if available). For example, at the first optimization level, the __debug__ constant becomes False and assert statements are optimized away, while at the second level, documentation strings are removed from the modules' bytecode.

In a PyInstaller-frozen applications, the optimization level of the embedded python interpreter is controlled by setting *python interpreter options* that are set at the build time. This affects the value of optimize flag in sys.flags. However, as PyInstaller by default collects pure-python modules in byte-compiled form, the value of the optimize flag at run time has no effect on the bytecode of such modules. I.e., even if optimization level of python interpreter in the frozen application is set to the second level via *python interpreter options*, assert statements will continue to work, and functions will retain their documentation strings. In order to affect the bytecode, the optimization level needs to be enforced during the build, specifically when PyInstaller compiles bytecode of modules that are to be collected.

In PyInstaller <= 6.5, the only way to affect optimization level of the collected python code was to set the optimization level of the python process in which PyInstaller was running; either by setting the PYTHONOPTIMIZE environment variable prior to running the PyInstaller command, or by invoking PyInstaller as a module and setting the python's -O flag, for example python -OO -m PyInstaller <...>.

In PyInstaller 6.6, an explicit bytecode optimization setting has been added both to the Analysis object in the *spec file* and to the command-line interface, in the form of the *--optimize* command-line option.

Optimization setting in the spec file

Starting with PyInstaller 6.6, the constructor of the Analysis object in the *spec file* accepts an integer parameter called optimize. This parameter directly controls the optimization level of bytecode for collected python modules and the program's entry-point script.

Setting the optimization level to a fixed value (0, 1, or 2) helps ensuring that the collected bytecode is always compiled with the specified optimization level, regardless of the optimization level under which the build process is running. On the other hand, setting the value to -1 will cause the bytecode optimization level to be inherited from the build process (the behavior of older PyInstaller versions).

Note that the optimize parameter passed to Analysis affects only the bytecode of collected modules. The run-time optimization level of the embedded interpreter (reflected in the value of optimize flag in sys.flags as shown at run-time) is still controlled by *python interpreter options* passed to the EXE constructor, and at the spec file level, the two settings are *not* coupled in any way.

Therefore, if you want to disable assert statements in the collected modules as well as ensure that sys.flags.optimize displays 1 at run time, you need to pass optimize=1 parameter to Analysis and pass a [('O', None, 'OPTION')] to EXE (as per *Specifying Python Interpreter Options*).

Using the --optimize command-line option

PyInstaller 6.6 introduced a new command-line option, called *--optimize*. This option can be used with pyi-makespec when generating a *spec file* for later use, or with pyinstaller when building directly from a .py file (i.e., with spec file generated on-the-fly during the build).

In the generated *spec file*, the value passed via *--optimize* option is passed to Analysis via the optimize argument, and in addition, the corresponding *python interpreter options* are also generated for the EXE. Therefore, this is the preferred approach to specifying the target bytecode optimization level for the frozen application.

If `--optimize` is not used on the command-line, but `--python-option` is used to pass the `O` *python interpreter options*, the optimization level is inferred from number of such options, and passed to `Analysis` in the generated spec file.

If neither `--optimize` nor `--python-option` are used, the optimization level for the generated spec file is determined from the optimization level of python interpreter under which PyInstaller is running. In the generated spec file, the inherited optimization level is passed to `Analysis` (thus fixing the optimization level for subsequent builds) and corresponding `OPTION` entries are generated for `EXE` as necessary.

Optimization level and the modulegraph's code-cache

During the import analysis process, PyInstaller's modulegraph ends up retrieving the code objects (bytecode) for all python modules that pass through analysis.

If the optimization level of the PyInstaller's build process matches the target optimization level for collected modules, the modulegraph's code-object cache can be reused, which helps to speed up the build process.

2.10.5 macOS multi-arch support

With the introduction of Apple Silicon M1, there are now several architecture options available for python:

- single-arch `x86_64` with thin binaries: older *python.org* builds, Homebrew python running natively on Intel Macs or under *rosetta2* on M1 Macs
- single-arch `arm64` with thin binaries: Homebrew python running natively on M1 macs
- multi-arch `universal2` with fat binaries (i.e., containing both `x86_64` and `arm64` slices): recent `universal2` *python.org* builds

PyInstaller aims to support all possible combinations stemming from the above options:

- single-arch application created using corresponding single-arch python
- `universal2` application created using `universal2` python
- single-arch application created using `universal2` python (i.e., reducing `universal2` fat binaries into either `x86_64` or `arm64` thin binaries)

By default, PyInstaller targets the current running architecture and produces a single-arch binary (`x86_64` when running on Intel Mac or under *rosetta2* on M1 Mac, or `arm64` when running on M1 Mac). The reason for that is that even with a `universal2` python environment, some packages may end up providing only single-arch binaries, making it impossible to create a functional `universal2` frozen application.

The alternative options, such as creating a `universal2` version of frozen application, or creating a non-native single-arch version using `universal2` environment, must therefore be explicitly enabled. This can be done either by specifying the target architecture in the `.spec` file via the `target_arch=` argument to `EXE()`, or on command-line via the `--target-arch` switch. Valid values are `x86_64`, `arm64`, and `universal2`.

Architecture validation during binary collection

To prevent run-time issues caused by missing or mismatched architecture slices in binaries, the binary collection process performs strict architecture validation. It checks whether collected binary files contain required arch slice(s), and if not, the build process is aborted with an error message about the problematic binary.

In such cases, creating frozen application for the selected target architecture will not be possible unless the problem of missing arch slices is manually addressed (for example, by downloading the wheel corresponding to the missing architecture, and stiching the offending binary files together using the `lipo` utility).

Changed in version 4.10: In earlier PyInstaller versions, the architecture validation was performed on all collected binaries, such as python extension modules and the shared libraries referenced by those extensions. As of PyInstaller 4.10, the architecture validation is limited to only python extension modules.

The individual architecture slices in a multi-arch `universal2` extension may be linked against (slices in) `universal2` shared libraries, or against distinct single-arch thin shared libraries. This latter case makes it impossible to reliably validate architecture of the collected shared libraries w.r.t. the target application architecture.

However, the extension modules do need to be fully compatible with the target application architecture. Therefore, their continued validation should hopefully suffice to detect attempts at using incompatible single-arch python packages*[0].

Trimming fat binaries for single-arch targets

When targeting a single architecture, the build process extracts the corresponding arch slice from any collected fat binaries, including the bootloader. This results in a completely thin build even when building in `universal2` python environment.

2.10.6 macOS binary code signing

With Apple Silicon M1 architecture, macOS introduced mandatory code signing, even if ad-hoc (i.e., without actual code-signing identity). This means that `arm64` arch slices (but possibly also `x86_64` ones, especially in `universal2` binaries) in collected binaries always come with signature.

The processing of binaries done by PyInstaller (e.g., library path rewriting in binaries' headers) invalidates their signatures. Therefore, the signatures need to be re-generated, otherwise the OS refuses to load a binary.

By default, PyInstaller ad-hoc (re)signs all collected binaries and the generated executable itself. Instead of ad-hoc signing, it is also possible to use real code-signing identity. To do so, either specify your identity in the `.spec` file via `codesign_identity=` argument to `EXE()` , or on command-line via the *`--codesign-identity`* switch.

Being able to provide codesign identity allows user to ensure that all collected binaries in either `onefile` or `onedir` build are signed with their identity. This is useful because for `onefile` builds, signing of embedded binaries cannot be performed in a post-processing step.

Note: When codesign identity is specified, PyInstaller also turns on *hardened runtime* by passing `--options=runtime` to the `codesign` command. This requires the codesign identity to be a valid Apple-issued code signing certificate, and will not work with self-signed certificates.

Trying to use self-signed certificate as a codesign identity will result in shared libraries failing to load, with the following reason reported:

> *[libname]: code signature in ([libname]) not valid for use in process using Library Validation: mapped file has no Team ID and is not a platform binary (signed with custom identity or adhoc?)*

[0] Although nothing really prevents a package from having distinct, architecture-specific extension modules…

Furthermore, it is possible to specify entitlements file to be used when signing the collected binaries and the executable. This can be done in the `.spec` file via `entitlements_file=` argument to `EXE()`, or on command-line via the *`--osx-entitlements-file`* switch.

App bundles

PyInstaller also automatically attempts to sign *.app bundles*, either using ad-hoc identity or actual signing identity, if provided via *`--codesign-identity`* switch. In addition to passing same options as when signing collected binaries (identity, hardened runtime, entitlement), deep signing is also enabled via by passing `--deep` option to the `codesign` utility.

Should the signing of the bundle fail for whatever reason, the error message from the `codesign` utility will be printed to the console, along with a warning that manual intervention and manual signing of the bundle are required.

2.10.7 macOS event forwarding and argv emulation in app bundles

The user interaction with macOS app bundles takes place via so called Apple Events. When the user double clicks on the application's icon, the application is started and receives an Open Application (`'oapp'`) event. Dragging a document on the application's icon or attempting to open an application-registered file generates an Open Document (`'odoc'`) event. Similarly, launching an URL with application-registered schema generates a Launch/Get URL (`'GURL'`) event. Typically, a long-running UI application installs `Carbon` or `Cocoa` event handlers (or their equivalents provided by higher-level UI toolkit) to handle these requests during its runtime.

PyInstaller provides two aspects of support for macOS event handling; automatic *event forwarding*, which enables frozen application to receive events in `onefile` mode, and optional *argv emulation* for converting initial opening event into `sys.argv` arguments. Both aspects apply only to app bundles (i.e., the `windowed` bootloader variant) and not to POSIX (command-line) frozen applications.

Changed in version 5.0: In earlier PyInstaller versions, *argv emulation* was always enabled in `onefile` mode and was unavailable in `onedir` mode. As PyInstaller 5.0, *argv emulation* must be explicitly opted-in, and is available in both `onefile` and `onedir` mode.

Event forwarding

In PyInstaller `onedir` bundles, the application runs as a single process, and therefore receives Apple Events normally, as other macOS applications would.

In `onefile` bundles, the application has a parent launcher process and the child process; the open document requests generated by user are received by the parent process, and are automatically forwarded to the child process, where the frozen python code is running.

Event forwarding is implemented for the following types of Apple Events:

- `kAEOpenDocuments` (`'odoc'`): open document request
- `kAEGetURL` (`'GURL'`): open/launch URL request
- `kAEReopenApplication` (`'rapp'`): reopen application
- `kAEActivate` (`'actv'`): activate application (bring to front)

Optional argv emulation

PyInstaller implements an optional feature called *argv emulation*, which can be toggled via `argv_emulation=` argument to `EXE()` in the *.spec file*, or enabled on command-line via `--argv-emulation` flag.

If enabled, the bootloader performs initial Apple Event handling to intercept events during the application's start-up sequence, and appends file paths or URLs received via Open Document/URL ('odoc' and 'GURL') events to `sys.argv`, as if they were received via command-line.

This feature is intended for simple applications that do not implement the event handling, but still wish to process initial open document request. This applies only to initial open events; events that occur after the frozen python code is started are dispatched via event queue (in `onedir` mode directly, and forwarded to child process in `onefile` mode.) and as such need to be handled via event handlers.

Note: This feature is not suitable for long-running applications that may need to service multiple open requests during their lifetime. Such applications will require proper event handling anyay, and therefore do not benefit from having initial events processed by *argv emulation*.

Warning: The initial event processing performed by bootloader in `onedir` mode may interfere with UI toolkit used by frozen python application, such as `Tcl/Tk` via `tkinter` module. The symptoms may range from window not being brought to front when the application startup to application crash with segmentation fault.

While PyInstaller tries to mitigate the issue on its end, we recommend against using *argv emulation* in combination with UI toolkits.

Practical examples

This section provides some practical examples on handling file and URL open events in macOS application bundles, via *argv emulation* in a simple one-shot program, or via installed event handlers in a GUI application.

Registering supported file types and custom URL schemas

In order for macOS application bundle to handle open operations on files and custom URL schemas, the OS needs to be informed what file types and what URL schemas the application supports. This is done in the bundle's `Info.plist` file, via `CFBundleDocumentTypes` and `CFBundleURLTypes` entries:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
↪PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  [...] <!-- preceding entries --->
  <key>CFBundleDocumentTypes</key>
  <array>
    <dict>
      <key>CFBundleTypeName</key>
      <string>MyCustomFileType</string>
      <key>CFBundleTypeExtensions</key>
      <array>
        <string>mcf</string>
```

(continues on next page)

(continued from previous page)

```
      </array>
      <key>CFBundleTypeRole</key>
      <string>Viewer</string>
    </dict>
  </array>
  <key>CFBundleURLTypes</key>
  <array>
    <dict>
      <key>CFBundleURLName</key>
      <string>MyCustomUrlSchema</string>
      <key>CFBundleTypeRole</key>
      <string>Viewer</string>
      <key>CFBundleURLSchemes</key>
      <array>
        <string>my-url</string>
      </array>
    </dict>
  </array>
</dict>
</plist>
```

In the above example, the application declares itself a viewer for made-up `.mcf` files, and as a viewer for URLs beginning with `my-url://`.

PyInstaller automatically generates an `Info.plist` file for your application bundle; to have it include the entries shown above, add the `info_plist` argument to the `BUNDLE()` directive in the *.spec file*, and set its content as follows:

```
app = BUNDLE(
    # [...]
    info_plist={
        'CFBundleURLTypes': [{
            'CFBundleURLName': 'MyCustomUrlSchema',
            'CFBundleTypeRole': 'Viewer',
            'CFBundleURLSchemes': ['my-url', ],
        }],
        'CFBundleDocumentTypes': [{
            'CFBundleTypeName': 'MyCustomFileType',
            'CFBundleTypeExtensions': ['mcf', ],
            'CFBundleTypeRole': "Viewer",
        }],
    }
)
```

Open event handling with argv emulation

Consider the following python script that began its life as a command-line utility, to be invoked from the terminal:

```
python3 img2gray.py image1.png image2.png ...
```

The script processes each passed image, converts it to grayscale, and saves it next to the original, with *-gray* appended to the file name:

```
# img2gray.py
import sys
import os

import PIL.Image


if len(sys.argv) < 2:
    print(f"Usage: {sys.argv[0]} <filename> [filenames...]")
    sys.exit(1)

# Convert all given files
for input_filename in sys.argv[1:]:
    filename, ext = os.path.splitext(input_filename)
    output_filename = filename + '-gray' + ext

    img = PIL.Image.open(input_filename)
    img_g = img.convert('L')
    img_g.save(output_filename)
```

If you generate an application bundle (as opposed to a command-line POSIX application), the most likely way of user interaction will be dragging image files onto the bundle's icon or using `Open with...` entry from the image file's context menu. Such interaction generates open file events, and in general requires your application code to implement event handling.

Enabling *argv emulation* in PyInstaller causes its bootloader to process events during the application startup, and extend `sys.argv` with any file paths or URLs that might have been received via open file or URL requests. This allows your application to process the received filenames as if they were passed via command-line, without any modifications to the code itself.

The following *.spec file* provides a complete example for a `onedir` application bundle that allows conversion of `.png` and `.jpg` images:

```
# img2gray.spec
a = Analysis(['img2gray.py'], )

pyz = PYZ(a.pure)

exe = EXE(
     pyz,
     a.scripts,
     exclude_binaries=True,
     name='img2gray',
     debug=False,
     bootloader_ignore_signals=False,
```

(continues on next page)

(continued from previous page)

```
    strip=False,
    upx=False,
    console=False,
    argv_emulation=True,  # enable argv emulation
)

coll = COLLECT(
    exe,
    a.binaries,
    a.datas,
    strip=False,
    upx=False,
    upx_exclude=[],
    name='img2gray'
)

app = BUNDLE(
    coll,
    name='img2gray.app',
    # Register .png and .jpg as supported file types
    info_plist={
        'CFBundleDocumentTypes': [{
            'CFBundleTypeName': "Convertible image types",
            'CFBundleTypeExtensions': [
                'png', 'jpg',
            ],
            'CFBundleTypeRole': "Viewer",
        }],
    }
)
```

The user can now drag image file(s) onto the icon of the resulting `img2gray` application bundle, or select `img2gray` under the `Open with...` entry in the image file's context menu.

Note: The *argv emulation* handles only initial open event, which is received before your frozen python code is started. If you wish to handle subsequent open requests while the application is still running, you need to implement proper event handling in your python code.

Open event handling in a `tkinter`-based GUI application

The Tcl/Tk framework used by `tkinter` allows application to provide event handlers for pre-defined types of Apple Events, by registering macOS-specific commands.

The handler for open file events can be registered via `::tk::mac::OpenDocument` command, while the handler for open URL events can be registered via `::tk::mac::LaunchURL` command. The latter is available starting with Tcl/Tk 8.6.10†[0].

[0] At the time of writing, python.org builds use Tcl/Tk 8.6.5, except for the Python 3.9.x *macOS 64-bit universal2 installer* builds, which use Tcl/Tk 8.6.10. Homebrew Python requires `tkinter` to be explicitly installed as `python-tk`, and uses latest version of Tcl/Tk, 8.6.11. Registering `::tk::mac::LaunchURL` command with versions of Tcl/Tk older than 8.6.10 is essentially no-op.

The following application illustrates the event handling using `tkinter`, by logging all received open file/URL events into a scrollable text widget:

```
# eventlogger_tk.py
import sys

import tkinter
import tkinter.scrolledtext


class Application:
    def __init__(self):
        # Create UI
        self.window = tkinter.Tk()
        self.window.geometry('800x600')
        self.window.title("Tk-based event logger")

        self.text_view = tkinter.scrolledtext.ScrolledText()
        self.text_view.pack(fill=tkinter.BOTH, expand=1)
        self.text_view.configure(state='disabled')

        # Register event handlers
        # See https://tcl.tk/man/tcl/TkCmd/tk_mac.html for list of
        # macOS-specific commands
        self.window.createcommand("::tk::mac::OpenDocument", self.open_document_handler)
        self.window.createcommand("::tk::mac::LaunchURL", self.open_url_handler)  #␣
↪works with Tcl/Tk >= 8.6.10

    def append_message(self, msg):
        """Append message to text view."""
        self.text_view.configure(state='normal')
        self.text_view.insert('end', msg + '\n')
        self.text_view.configure(state='disabled')

    def run(self):
        """Run the main loop."""
        app.append_message("Application started!")
        app.append_message(f"Args: {sys.argv[1:]}")
        self.window.mainloop()

    # Event handlers
    def open_document_handler(self, *args):
        app.append_message(f"Open document event: {args}")

    def open_url_handler(self, *args):
        app.append_message(f"Open URL event: {args}")


if __name__ == '__main__':
    app = Application()
    app.run()
```

The corresponding *.spec file* that builds a `onedir` application bundle with a custom file association (`.pyi_tk`) and a custom URL schema (`pyi-tk://`):

```
a = Analysis(['eventlogger_tk.py'])

pyz = PYZ(a.pure)

exe = EXE(
    pyz,
    a.scripts,
    exclude_binaries=True,
    name='eventlogger_tk',
    debug=False,
    bootloader_ignore_signals=False,
    strip=False,
    upx=False,
    console=False,
    argv_emulation=False,  # unnecessary as app handles events
)

coll = COLLECT(
    exe,
    a.binaries,
    a.datas,
    strip=False,
    upx=False,
    name='eventlogger_tk'
)

app = BUNDLE(
    coll,
    name='eventlogger_tk.app',
    # Register custom protocol handler and custom file extension
    info_plist={
        'CFBundleURLTypes': [{
            'CFBundleURLName': 'MyCustomUrlSchemaTk',
            'CFBundleTypeRole': 'Viewer',
            'CFBundleURLSchemes': ['pyi-tk'],
        }],
        'CFBundleDocumentTypes': [{
            'CFBundleTypeName': 'MyCustomFileTypeTk',
            'CFBundleTypeExtensions': [
                'pyi_tk',
            ],
            'CFBundleTypeRole': "Viewer",
         }],
    }
)
```

Once running, the application logs all received open file and open URL requests. These are generated either by trying to open a file with `.pyi_tk` extension using the UI, or using `open` command from the terminal:

```
$ touch file1.pyi_tk file2.pyi_tk file3.pyi_tk file4.pyi_tk

$ open file1.pyi_tk
$ open file2.pyi_tk
```

(continues on next page)

(continued from previous page)

```
$ open pyi-tk://test1
$ open pyi-tk://test2

$ open file3.pyi_tk file4.pyi_tk
```

Open event handling in a Qt-based GUI application

In Qt-based applications, open file and open URL requests are handled by installing application-wide event filter for QFileOpenEvent.

This event abstracts both open file and open URL request, with file open requests having `file://` URL schema. An event contains a single file name or URL, so an open request containing multiple targets generates corresponding number of `QFileOpenEvent` events.

Below is an example application and its corresponding *.spec file*:

```
# eventlogger_qt.py
import sys
import signal

from PySide2 import QtCore, QtWidgets


class Application(QtWidgets.QApplication):
    """
    QtWidgets.QApplication with extra handling for macOS Open
    document/URL events.
    """
    openFileRequest = QtCore.Signal(QtCore.QUrl, name='openFileRequest')

    def event(self, event):
        if event.type() == QtCore.QEvent.FileOpen:
            # Emit signal so that main window can handle the given URL.
            # Or open a new application window for the file, or whatever
            # is appropriate action for your application.
            self.openFileRequest.emit(event.url())
            return True
        return super().event(event)


class MainWindow(QtWidgets.QMainWindow):
    """
    Main window.
    """
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.resize(800, 600)

        self.setWindowTitle("Qt-based event logger")
```

(continues on next page)

(continued from previous page)

```
        # Construct the UI
        self.scroll_area = QtWidgets.QScrollArea()
        self.scroll_area.setWidgetResizable(True)
        self.setCentralWidget(self.scroll_area)

        self.text_edit = QtWidgets.QTextEdit()
        self.scroll_area.setWidget(self.text_edit)
        self.text_edit.setReadOnly(True)

    def append_message(self, msg):
        """
        Append message to text view.
        """
        self.text_edit.append(msg)

    def handle_open_file_request(self, url):
        self.append_message(f"Open request: {url.toString()}")


if __name__ == '__main__':
    # Make Ctrl+C work
    signal.signal(signal.SIGINT, signal.SIG_DFL)

    app = Application(list(sys.argv))

    window = MainWindow()
    window.show()

    window.append_message("Application started!")
    window.append_message(f"Args: {sys.argv[1:]}")

    app.openFileRequest.connect(window.handle_open_file_request)

    app.exec_()
```

```
# eventlogger_qt.spec
a = Analysis(['eventlogger_qt.py'])

pyz = PYZ(a.pure)

exe = EXE(
    pyz,
    a.scripts,
    exclude_binaries=True,
    name='eventlogger_qt',
    debug=False,
    bootloader_ignore_signals=False,
    strip=False,
    upx=False,
    console=False,
    argv_emulation=False,  # unnecessary as app handles events
)
```

(continues on next page)

(continued from previous page)

```
coll = COLLECT(
    exe,
    a.binaries,
    a.datas,
    strip=False,
    upx=False,
    name='eventlogger_qt'
)

app = BUNDLE(
    coll,
    name='eventlogger_qt.app',
    # Register custom protocol handler and custom file extension
    info_plist={
        'CFBundleURLTypes': [{
            'CFBundleURLName': 'MyCustomUrlSchemaQt',
            'CFBundleTypeRole': 'Viewer',
            'CFBundleURLSchemes': ['pyi-qt'],
        }],
        'CFBundleDocumentTypes': [{
            'CFBundleTypeName': 'MyCustomFileTypeQt',
            'CFBundleTypeExtensions': [
                'pyi_qt',
            ],
            'CFBundleTypeRole': "Viewer",
         }],
    }
)
```

The application behaves in the same way as its `tkinter`-based counterpart, except that the associated file extension and URL schema have been adjusted to prevent interference between the two example applications.

Initial open event

This section contains notes about behavior of the initial open event received by appliation, as seen by the frozen python code (or the UI toolkit it uses).

When application is opened normally, this is done via Open Application (`'oapp'`) event, which is the first event received by the application. If application is opened in response to open document or open URL request (i.e., it is not yet running when request is made), then the first received event is `'odoc'` or `'GURL'`, respectively.

In PyInstaller-frozen `onefile` bundles, the child process always starts with `'oapp'` event, regardless how the application was started. This is because the child is always started "normally", and it is the parent who receives the actual opening event; if the parent was opened with `'odoc'` or `'GURL'` event, then event is either forwarded to child or converted to `sys.argv` that is passed to the child, depending on whether *argv emulation* is enabled or not.

Therefore, in `onefile` mode, *argv emulation* has no direct effect on the initial open event (as seen by the frozen python code), which is always `'oapp'`.

In `onedir` bundles, there application consists of single process, which receives the events. Without *argv emulation*, the initial open event (as seen by the frozen python code) may be either `'oapp'`, `'odoc'`, or `'GURL'`, depending on how application was started.

However, if *argv emulation* is enabled in a `onedir` bundle, its processing of initial event leaves the event queue empty. The lack of initial open event seems to cause segmentation fault with Tcl/Tk 8.6.11 and Homebrew Python 3.9.6 (#5581). As a work-around, the bootloader attempts to submit an `'oapp'` event to itself, so that when the frozen python code inspects the event queue, it finds an initial open event (i.e., `'oapp'`). These potential side effects of *argv emulation* on UI toolkits are the reason why we recommend against using them together.

2.10.8 Signal handling in console Windows applications and onefile application cleanup

The signal handling in console applications on Windows differs from POSIX-based operating systems, such as linux and macOS. While signals generated by abnormal conditions, such as `SIGABRT` (abnormal termination; for example due to C code calling abort), `SIGFPE` (floating-point error), and `SIGSEGV` (illegal storage access), are generated and can be handled using handlers installed via the signal function, this is not the case for signals associated with program interruption and termination.

Specifically, interrupting a console-enabled program by pressing *Ctrl+C* does not generate the `SIGINT` signal, but rather a special *console control signal* called `CTRL_C_EVENT`, which can be handled by a handler installed via the SetConsoleCtrlHandler win32 API function[1]. Similarly, as noted in MSDN documentation on signal, the `SIGTERM` signal is not generated under Windows. Instead, there are several console control signals:

- `CTRL_C_EVENT`: interrupt via *Ctrl+C* key combination
- `CTRL_BREAK_EVENT`: interrupt via *Ctrl+Break* key combination
- `CTRL_CLOSE_EVENT`: closing the parent console window
- `CTRL_LOGOFF_EVENT`: a user logging off
- `CTRL_SHUTDOWN_EVENT`: system shutting down

When a console control signal is generated, the handler installed via SetConsoleCtrlHandler (if any) is executed *in a separate thread*, spawned within the program process by the operating system. In other words, the handler function is executed in parallel to the main program thread, which is necessary as the latter might be waiting on a blocking operation or performing an endless loop.

As noted here, upon receiving `CTRL_CLOSE_EVENT`, `CTRL_LOGOFF_EVENT`, or `CTRL_SHUTDOWN_EVENT`, the handler function can perform any necessary clean-up[2], and either:

- call ExitProcess to terminate the process.
- return `FALSE` (0). Other registered handlers are called, and if none returned `TRUE`, the default handler terminates the process by calling ExitProcess.
- return `TRUE` (non-zero). The system terminates the process immediately, without calling any other registered handler functions.

In other words, all options result in eventual program termination.

On the other hand, the default handler for `CTRL_C_EVENT` and `CTRL_BREAK_EVENT` also terminates the process, but this behavior can be modified by suppressing the default handler by returning `TRUE` in the user-installed one.

Another important aspect of console control signals is that handling `CTRL_CLOSE_EVENT`, `CTRL_LOGOFF_EVENT`, and `CTRL_SHUTDOWN_EVENT` is subject to system-imposed time-outs (e.g., five seconds for the `CTRL_CLOSE_EVENT`); if the process does not exit within the time-out limit, the operating system itself unconditionally terminates the process.

[1] The higher-level programming languages, such as python, might emulate the standard signals; but under-the-hood mechanics still involve console control signals discussed in this section.

[2] Note that at this point, however, the program is essentially a multi-threaded one, so usual multi-threading caveats may apply.

The above effectively means that once the program receives such control signal, its termination is inevitable (i.e., the signal cannot be ignored). At best, the termination can be delayed to perform any necessary clean-up, but even this must be done within system-imposed time limits.

Example of console control signal handling in python application

The following code demonstrates the basic implementation of a graceful console application shutdown. If the application is interrupted by user pressing *Ctrl+C* or *Ctrl+Break*, or closed due to user closing the console window, the application's state is stored to a file, so it can be restored on a subsequent run.

```
# console_counter.py
import sys
import time
import pathlib

import win32api  # pip install pywin32


def console_handler(signal):
    print(f"Console handler (signal {signal})!")
    global keep_running
    keep_running = False
    # Sleep until process either finishes or is killed by the OS
    time.sleep(20)
    return True


if __name__ == '__main__':
    keep_running = True

    # Install console handler
    win32api.SetConsoleCtrlHandler(console_handler, 1)

    # Restore state, if available
    state_file = pathlib.Path.home() / 'counter_state.txt'
    if state_file.is_file():
        print(f"Restoring state from {state_file}...", file=sys.stderr)
        try:
            with open(state_file, 'r') as fp:
                counter = int(fp.readline())
        except Exception:
            print("Failed to restore state from file!", file=sys.stderr)
            counter = 0
    else:
        print("State file does not exist!", file=sys.stderr)
        counter = 0

    print(f"Initial counter value: {counter}", file=sys.stderr)

    # Main loop
    while keep_running:
        print(f"Counter value: {counter}")
        counter += 1
```

(continues on next page)

(continued from previous page)

```
        time.sleep(1)

    # Clean-up
    print(f"Storing state to {state_file}...", file=sys.stderr)
    try:
        with open(state_file, 'w') as fp:
            print(f"{counter}", file=fp)
    except Exception:
        print(f"Failed to store state to {state_file}!", file=sys.stderr)

    print("Goodbye!")
    time.sleep(1)  # Delay exit for another second
```

The console control signal handler in the above code handles *all* console signals. This includes *Ctrl+C* event, which would otherwise generate a `KeyboardInterrupt` exception in the program's main thread[3]. After signalling the loop in the main thread to exit via the global boolean variable, the handler sleeps "forever". This approach works because the handler is executed in a separate thread, and this thread is terminated once the process ends - either due to main thread reaching its end, or due to the operating system terminating the process.

The above code should work as expected when executed as an unfrozen python script, and also when frozen by PyInstaller as a *onedir* application. However, *onefile* applications frozen with PyInstaller versions prior to 5.3 exhibit a problem; due to the lack of console control signals handling in the parent application process, the latter is always terminated immediately and leaves behind the unpacked temporary directory.

Changed in version 5.3: implemented handling of console control signals in the frozen application's parent process, which allows us to delay its termination until after the child process is terminated, and clean up the unpacked temporary directory. However, various caveats still apply, as discussed in the following sub-sections.

Onefile mode and temporary directory cleanup

The *onefile* mode in PyInstaller uses two processes. When the application is launched, the parent process extracts the contents of the embedded archive into a temporary directory, sets up the environment and library search paths, and launches the child process. The child process sets up the embedded python interpreter and runs the frozen python application. Meanwhile, the parent process waits for the child process to exit; when that happens, it cleans up the extracted temporary data, and exits.

From the perspective of the parent process, it does not matter whether the child process exits cleanly (i.e., with success code), or exits with an error code (for example, python code throws an exception that is not handled), or exits abnormally (e.g., crashes due to abnormal operation raising the `SIGABRT` signal), or is terminated by the OS (for example, from the Task Manager). In all cases, after the child process exits or is terminated, the parent process performs the cleanup, then exits with the exit code that was returned from the child process.

Therefore, in order for the application's temporary directory to be cleaned up, the parent process must never be forcefully terminated (for example, via the TerminateProcess function). If that happens, the clean-up code has no chance to run, and the temporary directory is left behind. On the other hand, from the perspective of the temporary directory clean-up, the child process can be terminated in any way, even forcefully. For the proper clean-up during a graceful shutdown triggered via console control signal (for example, due to *Ctrl+C* being pressed, or due to console window being closed), the bootloader in PyInstaller 5.3 and later attempts to delay the shut-down of the parent process so that the child process has time to exit and the main thread of the parent process has the chance to run the clean-up code.

The following sections provide additional details on this behavior for different situations.

[3] The `KeyboardInterrupt` exception could have been used to terminate the loop as well. However, that would not handle the `Ctrl+Break` key combination nor console window being closed.

Interrupting via Ctrl+C or Ctrl+Break

When *Ctrl+C* or *Ctrl+Break* is pressed in the console window, the `CTRL_C_EVENT` or `CTRL_BREAK_EVENT` is sent to all processes attached to that console[4].

In a *onefile* frozen application, the parent process ignores/suppresses the signal, so the outcome depends on how the frozen python code in the child process handles the signal. If the python code exits (for example, no handler is installed and `KeyboardInterrupt` exception interrupts the program flow), the parent process performs the clean-up and exits as well. If the python code in the child process handles the signal without shutting the child process down, the application keeps running.

This behavior is readily available in any PyInstaller version; in versions prior to 5.3, the parent process explicitly ignores `SIGABRT` and `SIGBREAK` signals, which achieves the same result as handling the corresponding console control signals, which is implemented from version 5.3 on.

Closing the console window

When the console window is closed (by pressing *X* button on title bar), the `CTRL_CLOSE_EVENT` is sent to all processes attached to that console[?].

In a *onefile* frozen application, the parent process receives the signal and suspends the handler's execution thread for 20 seconds. This way, the termination of the parent process is delayed, in order to give time to the child process (who also received the signal) to exit, and to the main thread of the parent process to perform cleanup and exit (which then also terminates the handler's execution thread). This behavior was implemented in PyInstaller 5.3 to ensure that closing the console window cleans up the application's temporary directory.

In versions prior to 5.3, the `CTRL_CLOSE_EVENT` is not handled; the parent process is terminated immediately without having the chance to perform the cleanup, leaving the application's temporary directory behind.

Note: The child process (i.e., the frozen python application code) might install its own console control signal handler in order to perform its own cleanup (for example, save the application's state). If so, it is important to keep in mind the system-imposed five-second timeout, and the fact that the parent process can perform the temporary directory cleanup only after the child process exits. In other words, if the clean up in the child process takes close to five seconds, the parent process may not have a chance to peform its own clean up before the OS kills the process.

Terminating the application via the Task Manager

Terminating the application via the Task Manager is somewhat unpredictable due to distinction between "Apps" and "Background processes".

"Apps" are closed by sending a close request to the application. Such applications may close gracefully if they close their window in response to the request, of, if they have a console, they handle the resulting `CTRL_CLOSE_EVENT` console control signal.

"Background processes" are terminated unconditionally using the TerminateProcess, leaving no hope for graceful shutdown and clean up.

The distinction between the two is based on whether the program has a visible window or not, but in practice, there are additional nuances when it comes to console-enabled applications and applications with multiple processes.

[4] If a *windowed/noconsole* application is started from a console, it is completely independent from it as long as it has a window. If the application has no window (i.e., a "hidden" application), its process does not receive `CTRL_C_EVENT` and `CTRL_BREAK_EVENT` signals in response to *Ctrl+C* and *Ctrl+Break* being pressed in the console, but is nevertheless terminated when the console is closed. The termination seems to be immediate and uncodnitional, i.e., without `CTRL_CLOSE_EVENT` signal being received.

To see the detailed classification on per-process basis, right click on the header of the process list view in the Task Manager, and enable display of the `Type` column. The newly added column will show the process classification for each process, and not just for the whole process group.

In the following sub-sections, we detail the behavior when attempting to shut down different processes involved with frozen applications. Roughly, the behavior higly depends on the following factors:

- build type: *onedir* (single-process) vs. *onefile* (two-process) PyInstaller build option.
- console enabled or not: *console* vs. *noconsole/windowed* PyInstaller build option.
- application has a window or not: regardless of whether an application has console enabled or not, it might have a window (window + console) or not (pure console-based application; or a "hidden", window-less and console-less, application that runs as a background process).
- how the application is launched: by double-clicking on the executable ("stand-alone", with its own console window) or by running it in an already-opened command prompt.

Windowed/noconsole onedir applications

Windowed/noconsole onedir applications are single-process applications without console, so they are the easiest to understand when it comes to the Task Manager and the shutdown behavior.

If the application has a window (for example, a Qt-based GUI), it is treated as an "App". It is listed under "Apps", and its process name is listed next to the top-level entry in the list. Shutting it down via the *"End task"* results in a window close event being posted, which allows for graceful application shutdown.

If the application has no window (a window-less and console-less "hidden" application), it is treated as a "Background process", and is listed under "Background processes". Shutting it down via the *"End task"* results in its unconditional termination, with no hope for graceful application shutdown.

As noted in earlier sections, *windowed/noconsole* applications are independent of the console even if they are launched from one, as long as they have a window. On the other hand, if an application has no window, the shutdown of the console process results in the immediate and uncoditional termination of the application process (background process within the console).

Because *onedir* applications do not need to unpack their contents to the temporary directory, the termination mode does not really affect the clean-up from PyInstaller's perspective. But it may be of concern if the application wishes to perform some clean-up on its own; for example, saving the current state during the shutdown as was done in *the earlier example*.

Console-enabled onedir applications

The shutdown behavior of Task Manager and *console-enabled onedir* applications depends on whether the application itself has a window (for example, a Qt-based GUI application with console enabled) or not (a "pure" console application), and whether the application owns the console window or not.

Pure console onedir application, ran via double-click

Running a pure-console application by double clicking on the executable opens a new console with the application running in it. The top-level entry in the process list is placed under "Apps"; however, it does not have a process name listed next to it. Instead, it is a group consisting of a *"Console Window Host"* (a "Windows process") and the actual application process, which is classified as an "App".

Shutting down the whole group (i.e., the top-level entry) via the *"End task"* results in everything being unconditionally terminated.

Shutting down the application process results in it receiving the `CTRL_CLOSE_EVENT` for graceful shutdown.

Pure console onedir application, ran in existing console

Opening a new command prompt results in a new *"Windows Command Processor"* group entry being added under "Apps". It consists of a *"Console Window Host"* (a "Windows process") and a *"Command Prompt"* (an "App"). Running a pure-console application from the opened command prompt results in a new process being added to the existing *"Windows Command Processor"* group, and the process is classified as a "Background process".

Therefore, shutting down the whole group results in everything being unconditionally terminated.

Shutting down the application process results in it being unconditionally terminated.

Shutting down the *"Command Prompt"* process results in application process receiving the `CTRL_CLOSE_EVENT` for graceful shutdown.

Console-enabled onedir application with window, ran via double-click

Running a console-enabled application with a window via double-click behaves similarly to the corresponding pure-console application case. The resulting process list entry is placed under "Apps", and is a group consisting of a *"Console Window Host"* (a "Windows process") and the actual application process, which is classified as an "App".

Shutting down the whole group results in everything being unconditionally terminated.

Shutting down the application process results in it receiving the `CTRL_CLOSE_EVENT` for graceful shutdown.

Console-enabled onedir application with window, ran in existing console

Running a console-enabled application with a window from an existing command prompt does not place the application process under the existing *"Windows Command Processor"* group, but rather results in a new "App" top-level entry in the process list. This entry behaves similarly to the `windowed onedir` case; it has process name listed nex to it and shutting it down via the *"End task"* results in a window close event being posted, which allows for graceful application shutdown.

Shutting down the whole *"Windows Command Processor"* closes the console, but the application itself keeps running (although its console handles likely become invalid[5]).

Shutting down the *"Command Prompt"* process within the *"Windows Command Processor"* group results in the application process receiving the `CTRL_CLOSE_EVENT` for graceful shutdown.

[5] Invalid console handles might, in turn, end up causing an error when the application code tries to use them, for example to print a message to the (now non-existent) console.

Console-enabled onefile applications

The shutdown behavior of *onefile* applications is complicated by the fact that two processes are involved, and that application contents need to be extracted to the temporary directory that should, ideally, be cleaned up when the application is shut down.

Pure-console onefile application, ran via double-click

Running a pure-console application by double clicking on the executable opens a new console with the application running in it. The top-level entry in the process list is placed under "Apps", and is a group consisting of:

- a *"Console Window Host"* (a "Windows process")
- the parent process, classified as an "App"
- the child process, classified as a "Background process"

Shutting down the whole group results in everything being unconditionally terminated. The temporary directory is left behind.

Shutting down the child process results in its immediate and unconditional termination. After the child process is terminated, the parent process performs temporary directory cleanup and exits, which also closes the console. The only potential drawback of this situation is that the application code cannot perform its own clean up.

Shutting down the parent process results in the `CTRL_CLOSE_EVENT` received by both parent and child process. After the child performs its cleanup (if any) and exits, the parent performs temporary directory cleanup and exits as well. This is the ideal situation[6].

Pure console onefile application, ran in existing console

Running a pure-console application from the opened command prompt results in two new processes being added to the existing *"Windows Command Processor"* group, and both of them are classified as a "Background process".

Shutting down the whole *"Windows Command Processor"* group results in everything being unconditionally terminated, and the temporary directory being left behind.

Shutting down the parent process results in its immediate and unconditional termination. The console accepts input again, while the child process (the actual application) keeps running in the background (i.e., still writing its output to the console). Since the parent process was terminated before it could perform clean-up, the temporary directory is left behind.

Shutting down the child process similarly results in its immediate and unconditional termination. After the child process is terminated, the parent process performs temporary directory cleanup and exits. The only potential drawback of this situation is that the application code cannot perform its own clean up.

Shutting down the *"Command Prompt"* process is the best choice, as it results in both the parent and the child process receiving the `CTRL_CLOSE_EVENT` for graceful shutdown.

But perhaps the most surefire way of closing the application in this case would be using *Ctrl+C* or *Ctrl+Break*, or even closing the console window.

[6] Assuming the potential cleanup in the application code does not delay the shutdown to the point where the OS ends up killing the parent process before it has the chance to perform the temporary directory cleanup…

Console-enabled onefile application with window, ran via double-click

Running a console-enabled application with a window via double-click results in two top-level entries in the process list.

The first entry is a group that belongs to the parent process; it contains a *"Console Window Host"* (a "Windows process") and the parent process, which is classified as an "App".

The child process is listed as a separate top-level entry that is also classified as an "App" and has process name listed next to it.

Shutting down the whole parent process group results in everything in that group being unconditionally terminated, while the child process (the actual application) keeps running. The temporary directory is left behind.

Shutting down the parent process results in the `CTRL_CLOSE_EVENT` received by both the parent and the child process. After the child performs its cleanup (if any) and exits, the parent performs temporary directory cleanup and exits as well. This is the ideal situation[?].

Shutting down the child process results in it receiving the `CTRL_CLOSE_EVENT` for graceful shutdown. After the child performs its cleanup (if any) and exits, the parent performs temporary directory cleanup and exits as well. This is the ideal situation; in this case, the parent process performs temporary directory cleanup even if the child process exceeds the signal handling timeout and is forcefully terminated by the operating system.

Console-enabled onefile application with window, ran in existing console

Running a console-enabled application with a window from the opened command prompt results in parent process being added to the existing *"Windows Command Processor"* group, as a "Background process".

The child process is listed as a separate top-level entry that is classified as an "App" and has process name listed next to it.

Shutting down the whole *"Windows Command Processor"* closes the console and results in immediate and unconditional termination of the parent process. The child process (the application itself) keeps running (although its console handles likely become invalid[?]). The temporary directory is left behind.

Shutting down the parent process results in its immediate and unconditional termination. The console is left open and accepts input again, while the child process (the actual application) keeps running in the background (i.e., still writing its output to the console). Since the parent process was terminated before it could perform clean-up, the temporary directory is left behind.

Shutting down the child process results in it receiving the `CTRL_CLOSE_EVENT` for graceful shutdown. After the child performs its cleanup (if any) and exits, the parent performs temporary directory cleanup and exits as well. This is the ideal situation; in this case, the parent process performs temporary directory cleanup even if the child process exceeds the signal handling timeout and is forcefully terminated by the operating system.

Shutting down the *"Command Prompt"* process results in both the parent and the child application process receiving the `CTRL_CLOSE_EVENT` for graceful shutdown. This is the ideal situation[?].

Windowed/noconsole onefile applications

In case of *windowed/noconsole onefile* applications, the application's parent process is usually classified as a "Background process". The classification of the child process depends on whether the application has a window or not.

Noconsole onefile application without window, ran via double-click

Running a "hidden" application (*noconsole/windowed* application without a window) by double clicking on the executable results in parent and child process being added to the process list as two distinct top-level entries, under "Background processes".

Shutting down the parent process results in its immediate and unconditional termination. The child process (the actual application) keeps running. Since the parent process was terminated before it could perform clean-up, the temporary directory is left behind.

Shutting down the child process also results in its immediate and unconditional termination. After the child process is terminated, the parent process performs temporary directory cleanup and exits. The only potential drawback of this situation is that the application code cannot perform its own clean up.

Noconsole onefile application without window, ran in existing console

Running a "hidden" application from the opened command prompt results in two new processes being added to the existing *"Windows Command Processor"* group, and both of them are classified as a "Background process".

Shutting down the whole *"Windows Command Processor"* group results in everything being unconditionally terminated, and the temporary directory being left behind.

Shutting down the parent process results in its immediate and unconditional termination. The child process (the actual application) keeps running as a background process. Since the parent process was terminated before it could perform clean-up, the temporary directory is left behind.

Shutting down the child process similarly results in its immediate and unconditional termination. After the child process is terminated, the parent process performs temporary directory cleanup and exits. The only potential drawback of this situation is that the application code cannot perform its own clean up.

Shutting down the *"Command Prompt"* process closes the console, but both parent and child process keep on running as background processes. Their entries are moved from the removed *"Windows Command Processor"* group into a new group entry under "Background processes".

Noconsole onefile application with window, ran via double-click

Running a regular GUI *noconsole* application via double click results in the parent process being classified as a "Background process" and the child process being classified as an "App". Each of them get their own top-level entry in the process list (under "Background processes" and under "Apps", respectively), and both have their process name listed next to them.

Shutting down the parent process results in its immediate and unconditional termination. The child process (the actual application) keeps running. Since the parent process was terminated before it could perform clean-up, the temporary directory is left behind.

Shutting down the child process results in a window close request (and the `CTRL_CLOSE_EVENT` signal) being sent to the child process for a graceful shutdown. After the child performs its cleanup (if any) and exits, the parent performs temporary directory cleanup and exits as well. This is the ideal situation; in this case, the parent process performs temporary directory cleanup even if the child process exceeds the signal handling timeout and is forcefully terminated by the operating system.

Noconsole onefile application with window, ran in existing console

Running a regular GUI *noconsole* application from an existing console is similar to running it via double-click, except that the parent process (classified as a "Background process") is listed under the *"Windows Command Processor"* group under "Apps" instead of a stand-alone entry under "Background processes".

Shutting down the whole *"Windows Command Processor"* closes the console and results in immediate and unconditional termination of the parent process. The child process (the application itself) keeps running. The temporary directory is left behind.

Shutting down the parent process results in its immediate and unconditional termination. This affects neither console nor the child process, both of which keep running. Since the parent process was terminated before it could perform clean-up, the temporary directory is left behind.

Shutting down the child process results in it receiving the `CTRL_CLOSE_EVENT` for graceful shutdown. After the child performs its cleanup (if any) and exits, the parent performs temporary directory cleanup and exits as well. This is the ideal situation; in this case, the parent process performs temporary directory cleanup even if the child process exceeds the signal handling timeout and is forcefully terminated by the operating system.

Shutting down the *"Command Prompt"* process results in console being closed and the parent process being immediately and unconditionally terminated. The child process keeps running. Since the parent process was terminated before it could perform clean-up, the temporary directory is left behind.

2.10.9 Automatic hiding and minimization of console window under Windows

For console-enabled Windows applications, PyInstaller offers an option to automatically hide or minimize the console window *when the console window is owned by the program's process* (i.e., the program was not launched from an existing console window).

Automatic minimization of console window allows a GUI application to put the console out of the user's way, while allowing it to be brought back if required. Automatic hiding of console window might be used to create an illusion of a hybrid application that has no console when launched by double-clicking on the executable, but shows console output when launched from existing console window.

Note that the programmatic hiding/minimization of console can be easily implemented by application itself using win32 API via `ctypes`. The advantage of having it in PyInstaller's bootloader is that:

- it can be performed very early in the program's life cycle (especially in case of `onefile` builds).
- in `onefile` builds, the bootloader can easily determine the ownership of console, regardless of parent and child process being used (as the check is executed in the parent process).

Also note that console hiding is different from `windowed`/`noconsole` builds, which have no console at all. This option works only with console-enabled builds, and involves PyInstaller's bootloader programmatically hiding or minimizing the console.

To enable this functionality, use the *--hide-console* command-line option, or corresponding `hide_console` argument to `EXE` in the .spec file. Currently, four modes are supported: `hide-early`, `minimize-early`, `hide-late`, and `minimize-late`.

Depending on the setting, the console is hidden/mininized either early in the bootloader execution or late in the bootloader execution. The early option takes place as soon as the PKG archive is found. In `onefile` builds, the late option takes place after application has unpacked itself and before it launches the child process. In `onedir` builds, the late option takes place before starting the embedded python interpreter.

Note: Even with hiding/minimizing console early in the bootloader's execution, the user might see console being opened for an instant before it is hidden or minimized.

In fact, hiding console before the application's UI is brought up might give the user an impression that the application has crashed. Therefore, it might be preferable to have the application code to implement its own programmatic hiding/minimization of the console window, and have it performed only after the UI becomes visible.

2.11 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.

2.11.1 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 PyQt5 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!

2.11.2 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 3.8". 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 macOS.

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 sent to any attached debugger. If you are not using a debugger (or don't have one), the DebugView the free (beer) tool can be used to display such messages. It has to be started before running the bundled application.

For a `--windowed` macOS app they are not displayed.

Consider bundling without `--debug` for your production version. Debugging messages require system calls and have an impact on performance.

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 may 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 macOS, 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 in 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/
```

2.11.3 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 in the `pathex` argument. 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__()`, `importlib.import_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.shell` module actually resolves to `win32com/win32comext/shell/shell.pyd`. This is because `win32com/__init__.py` appends `../win32comext` to its `__path__`.

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

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

Once in a while, that's not enough.

Changing Runtime Behavior

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

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

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

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

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

Hooks called in this way, while they need to be careful of what they import, are free to do almost anything. One reason to write a run-time hook is to override some functions or variables from some modules. A good example of this is the Django runtime hook (see `loader/rthooks/pyi_rth_django.py` in the PyInstaller folder). Django imports some

modules dynamically and it is looking for some `.py` files. However `.py` files are not available in the one-file bundle. We need to override the function `django.core.management.find_commands` in a way that will just return a list of values. The runtime hook does this as follows:

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

2.11.4 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
```

2.11.5 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.

2.12 Advanced Topics

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

2.12.1 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 *temppath*`/_MEI`*xxxxxx*.
   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 *temppath*`/_MEI`*xxxxxx*.

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.
3. Initialize Python interpreter: set sys.path, sys.prefix, sys.executable.
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`.

Splash screen startup

Note: This feature is incompatible with macOS. In the current design, the splash screen operates in a secondary thread, which is disallowed by the Tcl/Tk (or rather, the underlying GUI toolkit) on macOS.

If a splash screen is bundled with the application the bootloaders startup procedure and threading model is a little more complex. The following describes the order of operation if a splash screen is bundled:

1. The bootloader checks if it runs as the outermost application (Not the child process which was spawned by the bootloader).
2. If splash screen resources are bundled, try to extract them (onefile mode). The extraction path is inside *temppath*`/_MEI`*xxxxxx*`/__splash`*x*. If in onedir mode, the application assumes the resources are relative to the executable.
3. Load the tcl and tk shared libraries into the bootloader.
   - Windows: `tcl86t.dll/tk86t.dll`
   - Linux: `libtcl.so/libtk.so`
4. Prepare a minimal environment for the Tcl/Tk interpreter by replacing/modifying the following functions:
   1. `::tclInit`: This command is called to find the standard library of tcl. We replace this command to force tcl to load/execute only the bundled modules.
   2. `::tcl_findLibrary`: Tk uses this function to source all its components. The overwritten function sets the required environment variable and evaluates the requested file.
   3. `::exit`: This function is modified to ensure a proper exit of the splash screen thread.
   4. `::source`: This command executes the contents of a passed file. Since we run in a minimal environment we mock the execution of not bundled files and execute those who are.
5. Start the tcl interpreter and execute the splash screen script which was generated by PyInstaller's build target `Splash` at build time. This script creates the environment variable `_PYIBoot_SPLASH`, which is also available to the python interpreter. It also initializes a tcp server socket to receive commands from python.

Note: The tcl interpreter is started in a separate thread. Only after the tcl interpreter has executed the splash screen script, the bootloader thread, which is responsible for extraction/starting the python interpreter, is resumed.

2.12.2 `pyi_splash` Module (Detailed)

This module connects to the bootloader to send messages to the splash screen.

It is intended to act as an RPC interface for the functions provided by the bootloader, such as displaying text or closing. This makes the users python program independent of how the communication with the bootloader is implemented, since a consistent API is provided.

To connect to the bootloader, it connects to a local tcp server socket whose port is passed through the environment variable `_PYIBoot_SPLASH`. The bootloader connects to the socket via the python module `_socket`. Although this socket is bidirectional, the module is only configured to send data. Since the os-module, which is needed to request the environment variable, is not available at boot time, the module does not establish the connection until initialization.

This module does not support reloads while the splash screen is displayed, i.e. it cannot be reloaded (such as by `importlib.reload()`), because the splash screen closes automatically when the connection to this instance of the module is lost.

Functions

Note: Note that if the `_PYIBoot_SPLASH` environment variable does not exist or an error occurs during the connection, the module will **not** raise an error, but simply not initialize itself (i.e. *`pyi_splash.is_alive()`* will return `False`). Before sending commands to the splash screen, one should check if the module was initialized correctly, otherwise a `RuntimeError` will be raised.

`is_alive`()
Indicates whether the module can be used.

Returns `False` if the module is either not initialized or was disabled by closing the splash screen. Otherwise, the module should be usable.

`update_text`(*msg*)
Updates the text on the splash screen window.

Parameters **`msg`** (*`str`*) – the text to be displayed

Raises

- **`ConnectionError`** – If the OS fails to write to the socket
- **`RuntimeError`** – If the module is not initialized

`close`()
Close the connection to the ipc tcp server socket

This will close the splash screen and renders this module unusable. After this function is called, no connection can be opened to the splash screen again and all functions if this module become unusable

2.12.3 The Table of Contents (TOC) lists and the Tree Class

PyInstaller manages lists of files that are to be collected in the so-called Table of Contents (TOC) list format. These lists contain three-element tuples that encapsulate information about a file's destination name, the file's full source path, and its type.

As part of utilities for managing the TOC lists, PyInstaller provides a `Tree` class as a convenient way to build a TOC list from the contents of the given directory. This utility class can be used either in the *.spec files* file or from custom hooks.

Table of Contents (TOC) lists

The `Analysis` object produces several TOC lists that provide information about files to be collected. The files are grouped into distinct lists based on their type or function, for example: - `Analysis.scripts`: program script(s) - `Analysis.pure`: pure-python modules - `Analysis.binaries`: binary extension modules and shared libraries - `Analysis.datas`: data files

The generated TOC lists are passed to various build targets within the *spec file*, such as `PYZ`, `EXE`, and `COLLECT`.

Each TOC list contains three-element tuples,

```
(dest_name, src_name , typecode)
```

where `dest_name` is the destination file name (i.e., file name within the frozen application; as such, it must always be a relative name), `src_name` is the source file name (the path from where the file is collected), and `typecode` is a string that denotes the type of the file (or entry).

Internally, PyInstaller uses a number of *typecode* values, but for the normal case you need to know only these:

typecode	description	dest_name	src_name
'DATA'	Arbitrary (data) files.	Name in the frozen application.	Full path to the file on the build system.
'BINARY'	A shared library.	Name in the frozen application.	Full path to the file on the build system.
'EXTENSION'	A Python binary extension.	Name in the frozen application.	Full path to the file on the build system.
'OPTION'	A PyInstaller/Python runtime option.	Option name (and optional value, separated by a whitespace).	Ignored.

The destination name corresponds to the name of the final in the frozen application, relative to the top-level application directory. It may include path elements, for example `extras/mydata.txt`.

Entries of type `BINARY` and `EXTENSION` are assumed to represent a file containing loadable executable code, such as a dynamic library. Generally, `EXTENSION` is used to denote Python extensions modules, such as modules compiled by Cython. The two file types are treated in the same way; PyInstaller scans them for additional link-time dependencies and collects any dependencies that are discovered. On some operating systems, binaries and extensions undergo additional processing (such as path rewriting for link-time dependencies and code-signing on macOS).

The TOC lists produced by `Analysis` can be modified in the *spec file* file before they are passed on to the build targets to either include additional entries (although it is preferable to pass extra files to be included via *binaries* or *datas* arguments of *Analysis*) or remove unwanted entries.

Changed in version 5.11: In PyInstaller versions prior to 5.11, the TOC lists were in fact instances of the `TOC` class, which internally performed implicit entry de-duplication; i.e., trying to insert an entry with existing target name would result in no changes to the list.

However, due to the shortcomings of the `TOC` class that resulted from loosely-defined and conflicting semantics, the use of the `TOC` class has been deprecated. The TOC lists are now instances of plain `list`, and PyInstaller performs explicit list normalization (entry de-duplication). The explicit normalization is performed at the end of `Analysis` instantiation, when the lists are stored in the class' properties (such as `Analysis.datas` and `Analysis.binaries`). Similarly, explicit list normalization is also performed once the build targets (`EXE`, `PYZ`, `PKG`, `COLLECT`, `BUNDLE`) consolidate the input TOC lists into the final list.

The Tree Class

The `Tree` class offers a convenient way of creating a TOC list that describes contents of the given directory:

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

- The *root* argument is a string denoting the path to the directory. It may be absolute or relative to the spec file directory.
- The optional *prefix* argument is a name for a sub-directory in the application directory into which files are to be collected. If not specified or set to `None`, the files will be collected into the top-level application directory.
- The optional *excludes* argument 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

  - a glob pattern (e.g., `*.ext`), which causes matching files to be excluded

- The optional *typecode* argument specifies the TOC typecode string that is assigned to all entries in the TOC list. The default value is `DATA`, which is appropriate for most cases.

For example:

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

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

- A *dest_name* in form of:file:*extras/{filename}*.
- A *src_name* that corresponds to the full absolute path to that file in the `../src/extras` folder (relative to the location of the spec file).
- A *typecode* of `DATA` (the default).

An example of creating a TOC listing some binary modules:

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

This creates a TOC list with entries for each file in the `cy_mods` directory, excluding files with the `.pyx`, `.py`, or `.pyc` extension (so presumably collecting only the `.pyd` or `.so` modules created by Cython). Each tuple in this TOC has:

- A *dest_name* that corresponds to the file's basename (all files are collected in top-level application directory).
- A *src_name* that corresponds to the full absolute path to that file in `../src/cy_mods` relative to the spec file.
- A *typecode* of `EXTENSION` (`BINARY` could be used as well).

2.12.4 Inspecting Archives

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

ZlibArchive

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

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

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

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

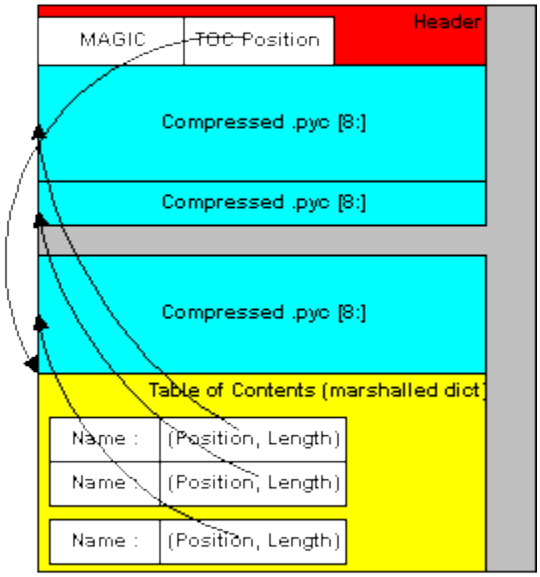

Fig. 1: Structure of the ZlibArchive

CArchive

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

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

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

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

The ELF executable format (Windows, 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*

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).

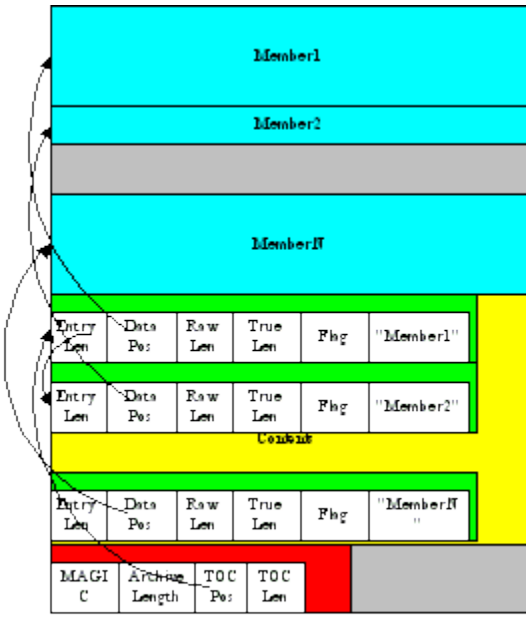

Fig. 2: Structure of the CArchive

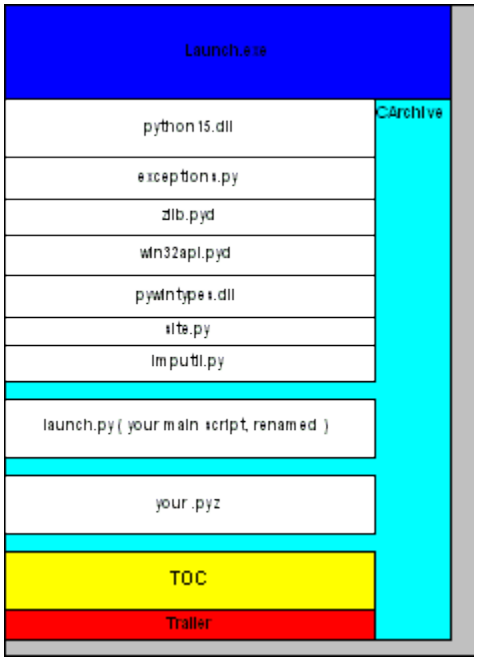

Fig. 3: Structure of the Self Extracting Executable

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.

2.12.5 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.

2.12.6 Creating a Reproducible Build

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

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

You can ensure 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 macOS):

```
# 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
```

Changed in version 4.8: The build timestamp in the PE headers of the generated Windows executables is set to the current time during the assembly process. A custom timestamp value can be specified via the `SOURCE_DATE_EPOCH` environment variable to achieve reproducible builds.

2.13 Understanding PyInstaller Hooks

Note: We strongly encourage package developers to provide hooks with their packages. See section *Providing PyInstaller Hooks with your Package* for how easy this is.

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 module. For example, `hook-PyQt5.QtCore.py` is a hook file corresponding to the module `PyQt5.QtCore`. When your script (or one of its dependencies) contains `import PyQt5.QtCore` (or `from PyQt5 import QtCore`), Analysis notes that `hook-PyQt5.QtCore.py` exists, and will call it.

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. Additional hooks are provided by the `pyinstaller-hooks-contrib` package, which is typically installed as part of PyInstaller dependencies. See here to browse PyInstaller-provided hooks in the online repository, and here for hooks provided by the `pyinstaller-hooks-contrib`.

Many hooks consist of only one statement, an assignment to `hiddenimports`. For example, the `xml.dom` module from Python standard library imports a module called `xml.dom.domreg`, which in turn indirectly imports `xml.dom.minidom` as one of registered XML DOM implementations. Therefore, to ensure that this implementation module is collected, PyInstaller provides a hook called `hook-xml.dom.domreg.py`, which contains only the following statement:

```
hiddenimports = ["xml.dom.minidom"]
```

When Analysis sees an `import xml.dom` statement in the user code (or one of its dependencies), and subsequently sees that `xml.dom` module imports the `xml.dom.domreg` module (via the `from .domreg import getDOMImplementation, registerDOMImplementation` statement), it calls `hook-xml.dom.domreg.py`, and examines the value of `hiddenimports` hook global variable set by the hook. As a result, the `xml.dom.minidom` module is collected into the frozen application, as if the `xml.dom.domreg` module (or your source script) contained a direct `import xml.dom.minidom` statement.

A hook can also cause the collection of data files or binaries (shared libraries) from a package, collection of metadata for a package, and it can also prevent collection of packages/modules that are imported only from the hooked module or a package. Examples of these actions are shown below.

When the module that needs a hook is useful only to your project, you can 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 ask the package developer to *include the hook with her/his package* or send us the hook file so we can include it in the contributed hooks repository.

2.13.1 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 many hooks in the PyInstaller installation, but a much larger collection can be found in the community hooks package. Please browse through them for examples.

2.13.2 Providing PyInstaller Hooks with your Package

As a package developer you can provide hooks for PyInstaller within your package. This has the major benefit that you can easily adopt the hooks when your package changes. Thus your package's users don't need to wait until PyInstaller might catch up with these changes. If both PyInstaller and your package provide hooks for some module, your package's hooks take precedence, but can still be overridden by the command line option *--additional-hooks-dir*.

You can tell PyInstaller about the additional hooks by defining some simple setuptools entry-points in your package. Therefore add entries like these to your `setup.cfg`:

```
[options.entry_points]
pyinstaller40 =
  hook-dirs = pyi_hooksample.__pyinstaller:get_hook_dirs
  tests     = pyi_hooksample.__pyinstaller:get_PyInstaller_tests
```

This defines two entry-points:

`pyinstaller40.hook-dirs` for hook registration This entry point refers to a function that will be invoked with no parameters. It must return a sequence of strings, each element of which provides an additional absolute path to search for hooks. This is equivalent to passing the *--additional-hooks-dir* command-line option to PyInstaller for each string in the sequence.

In this example, the function is `get_hook_dirs() -> List[str]`.

`pyinstaller40.tests` for test registration This entry point refers to a function that will be invoked with no parameters. It must return a sequence of strings, each element of which provides an additional absolute path to a directory tree or to a Python source file. These paths are then passed to *pytest* for test discovery. This allows both testing by this package and by PyInstaller.

In this project, the function is `get_PyInstaller_tests() -> List[str]`.

A sample project providing a guide for integrating PyInstaller hooks and tests into a package is available at https://github.com/pyinstaller/hooksample. This project demonstrates defining a library which includes PyInstaller hooks along with tests for those hooks and sample file for integration into CD/CI testing. Detailed documentation about this sample project is available at https://pyinstaller-sample-hook.readthedocs.io/en/latest/.

2.13.3 Hook Global Variables

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

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

```
hiddenimports = ['_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:

```
excludedimports = ['tkinter']
```

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

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

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

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

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

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

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

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

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

warn_on_missing_hiddenimports A boolean flag indicating whether missing hidden imports from the hook (set via `hiddenimports`) should generate warnings or not. By default, missing hidden imports generate warnings, but individual hooks can opt out of this behavior by setting this variable to `False`. Example:

```
warn_on_missing_hiddenimports = False
```

module_collection_mode A setting controlling the collection mode for module(s). The value can be either a string or a dictionary.

When set to a string, the variable controls the collection mode for the hooked package/module. Valid values are:

- `'pyz'`: collect byte-compiled modules into the embedded PYZ archive. This is the default behavior when no collection mode is specified. If the `noarchive` flag is used with `Analysis`, the PYZ archive is not used, and `pyz` collection mode is automatically turned into `pyc` one.
- `'pyc'`: collect byte-compiled modules as external data files (as opposed to collecting them into the PYZ archive).
- `'py'`: collect source .py files as external data files. Do not collect byte-compiled modules.
- `'pyz+py'` or `'py+pyz'`: collect byte-compiled modules into the embedded PYZ archive and collect corresponding source .py files as external data files.

  If `noarchive` flag is in effect, the byte-compiled modules are collected as external data files, which causes python to ignore them due to the source files being placed next to them.

The setting is applied to all child modules and subpackages, unless overridden by the setting in their corresponding hook.

Alternatively, the variable can be set to a dictionary comprising module/package names and corresponding collection mode strings. This allows a hook to specify different settings for its main package and subpackages, but also settings for other packages. When multiple hooks provide a setting for the same module name, the end result depends on the hook execution order.

Example:

```
# hook-mypackage.py

# This package must be collected in source form, due to its code
# searching for .py files on the filesystem...
module_collection_mode = 'py'
```

Example:

```
# hook-mypackage.py

# Collect only a sub-package / module as source
# (without creating a hook for the sub-package).
module_collection_mode = {
    'mypackage.src_subpackage': 'py'
}
```

Example:

```
# hook-mypackage.py

# Collect whole package as source except for a single sub-package
# (without creating a hook for the sub-package).
module_collection_mode = {
    'mypackage': 'py',
    'mypackage.bin_subpackage': 'pyz'
}
```

Example:

```
# hook-mypackage.py

# Force collection of other packages in source form.
```

(continues on next page)

(continued from previous page)

```
module_collection_mode = {
    'myotherpackage1': 'py',
    'myotherpackage2': 'py',
}
```

The ability to control collection mode for other modules/packages from a given hook is intended for cases when the hooked module provides functionality for other modules that requires those other modules to be collected in the source form (for example, JIT compilation available in some deep learning frameworks). However, detection of specific function imports and calls via bytecode scanning requires an access to the modulegraph, and consequently the use of the *the hook(hook_api) function*. In such cases, the collection mode can be modified using the *set_module_collection_mode method* from the `hook_api` object instead of setting the global hook variable.

2.13.4 Useful Items in `PyInstaller.compat`

Various classes and functions to provide some backwards-compatibility with previous versions of Python onward.

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

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

is_py36, is_py37, is_py38, is_py39, is_py310 is_py311
: True when the current version of Python is at least 3.6, 3.7, 3.8, 3.9, or 3.10, 3.11 respectively.

is_win
: True in a Windows system.

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

is_darwin
: True in macOS.

is_linux
: True in any GNU/Linux system.

is_solar
: True in Solaris.

is_aix
: True in AIX.

is_freebsd
: True in FreeBSD.

is_openbsd
: True in OpenBSD.

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.

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

2.13.5 Useful Items in `PyInstaller.utils.hooks`

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

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

The functions listed here are generally useful and used in a number of existing hooks.

exec_statement(*statement*)

Execute a single Python statement in an externally-spawned interpreter, and return the resulting standard output as a string.

Examples:

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

mpl_data_dir = exec_statement("import matplotlib; print(matplotlib.get_data_path())
↪")
datas = [ (mpl_data_dir, "") ]
```

Notes

As of v5.0, usage of this function is discouraged in favour of the new *PyInstaller.isolated* module.

eval_statement(*statement*)

Execute a single Python statement in an externally-spawned interpreter, and `eval()` its output (if any).

Example:

```
databases = eval_statement('''
   import sqlalchemy.databases
   print(sqlalchemy.databases.__all__)
   ''')
for db in databases:
   hiddenimports.append("sqlalchemy.databases." + db)
```

Notes

As of v5.0, usage of this function is discouraged in favour of the new *PyInstaller.isolated* module.

check_requirement(*requirement*)

Check if a **PEP 0508** requirement is satisfied. Usually used to check if a package distribution is installed, or if it is installed and satisfies the specified version requirement.

Parameters **requirement** (*str*) – Requirement string in **PEP 0508** format.

Returns Boolean indicating whether the requirement is satisfied or not.

Return type bool

Examples

```
# Assume Pillow 10.0.0 is installed.
>>> from PyInstaller.utils.hooks import check_requirement
>>> check_requirement('Pillow')
True
>>> check_requirement('Pillow < 9.0')
False
>>> check_requirement('Pillow >= 9.0, < 11.0')
True
```

is_module_satisfies(*requirements*, *version=None*, *version_attr=None*)

A compatibility wrapper for *check_requirement()*, intended for backwards compatibility with existing hooks.

In contrast to original implementation from PyInstaller < 6, this implementation only checks the specified **PEP 0508** requirement string; i.e., it tries to retrieve the distribution metadata, and compare its version against optional version specifier(s). It does not attempt to fall back to checking the module's version attribute, nor does it support `version` and `version_attr` arguments.

Parameters

- **requirements** (*str*) – Requirements string passed to the *check_requirement()*.
- **version** (*None*) – Deprecated and unsupported. Must be `None`.
- **version_attr** (*None*) – Deprecated and unsupported. Must be `None`.

Returns Boolean indicating whether the requirement is satisfied or not.

Return type bool

Raises **ValueError** – If either `version` or `version_attr` are specified and are not None.

collect_all(*package_name*, *include_py_files=True*, *filter_submodules=<function <lambda>>*, *exclude_datas=None*, *include_datas=None*, *on_error='warn once'*)

Collect everything for a given package name.

Parameters

- **package_name** (str) – An `import`-able package name.
- **include_py_files** (bool) – Forwarded to *collect_data_files()*.
- **filter_submodules** (Callable) – Forwarded to *collect_submodules()*.
- **exclude_datas** (UnionType[list, None]) – Forwarded to *collect_data_files()*.
- **include_datas** (UnionType[list, None]) – Forwarded to *collect_data_files()*.
- **on_error** (str) – Forwarded onto *collect_submodules()*.

Returns

A `(datas, binaries, hiddenimports)` triplet containing:

- All data files, raw Python files (if **include_py_files**), and distribution metadata directories (if applicable).
- All dynamic libraries as returned by *collect_dynamic_libs()*.
- All submodules of **package_name**.

Return type tuple

Typical use:

```
datas, binaries, hiddenimports = collect_all('my_package_name')
```

collect_submodules(*package*, *filter=<function <lambda>>*, *on_error='warn once'*)
List all submodules of a given package.

Parameters

- **package** (str) – An import-able package.
- **filter** (Callable[[str], bool]) – Filter the submodules found: A callable that takes a submodule name and returns True if it should be included.
- **on_error** (str) – The action to take when a submodule fails to import. May be any of:
  - raise: Errors are reraised and terminate the build.
  - warn: Errors are downgraded to warnings.
  - warn once: The first error issues a warning but all subsequent errors are ignored to minimise *stderr pollution*. This is the default.
  - ignore: Skip all errors. Don't warn about anything.

Returns All submodules to be assigned to hiddenimports in a hook.

This function is intended to be used by hook scripts, not by main PyInstaller code.

Examples:

```
# Collect all submodules of Sphinx don't contain the word ``test``.
hiddenimports = collect_submodules(
    "Sphinx", ``filter=lambda name: 'test' not in name)
```

Changed in version 4.5: Add the **on_error** parameter.

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.

Examples

The following excludes foo.test and foo.test.one but not foo.testifier.

```
collect_submodules('foo', lambda name: not is_module_or_submodule(name, 'foo.test
↪'))``
```

is_package(*module_name*)
Check if a Python module is really a module or is a package containing other modules, without importing anything in the main process.

Parameters **module_name** (str) – Module name to check.

Returns True if module is a package else otherwise.

collect_data_files(*package*, *include_py_files=False*, *subdir=None*, *excludes=None*, *includes=None*)
This function produces a list of (source, dest) entries for data files that reside in package. Its output can be directly assigned to datas in a hook script; for example, see hook-sphinx.py. The data files are all files that are not shared libraries / binary python extensions (based on extension check) and are not python source (.py) files or byte-compiled modules (.pyc). Collection of the .py and .pyc files can be toggled via the include_py_files flag. Parameters:

- The `package` parameter is a string which names the package.
- By default, python source files and byte-compiled modules (files with `.py` and `.pyc` suffix) are not collected; setting the `include_py_files` argument to `True` collects these files as well. This is typically used when a package requires source .py files to be available; for example, JIT compilation used in deep-learning frameworks, code that requires access to .py files (for example, to check their date), or code that tries to extend *sys.path* with subpackage paths in a way that is incompatible with PyInstaller's frozen importer.. However, in contemporary PyInstaller versions, the preferred way of collecting source .py files is by using the **module collection mode** setting (which enables collection of source .py files in addition to or in lieu of collecting byte-compiled modules into PYZ archive).
- The `subdir` argument gives a subdirectory relative to `package` to search, which is helpful when submodules are imported at run-time from a directory lacking `__init__.py`.
- The `excludes` argument contains a sequence of strings or Paths. These provide a list of globs to exclude from the collected data files; if a directory matches the provided glob, all files it contains will be excluded as well. All elements must be relative paths, which are relative to the provided package's path (/ `subdir` if provided).

  Therefore, `*.txt` will exclude only `.txt` files in `package`'s path, while `**/*.txt` will exclude all `.txt` files in `package`'s path and all its subdirectories. Likewise, `**/__pycache__` will exclude all files contained in any subdirectory named `__pycache__`.
- The `includes` function like `excludes`, but only include matching paths. `excludes` override `includes`: a file or directory in both lists will be excluded.

This function does not work on zipped Python eggs.

This function is intended to be used by hook scripts, not by main PyInstaller code.

`collect_dynamic_libs`(*package*, *destdir=None*, *search_patterns=['*.dll', '*.dylib', 'lib*.so']*)

This function produces a list of (source, dest) of dynamic library files that reside in package. Its output can be directly assigned to `binaries` in a hook script. The package parameter must be a string which names the package.

Parameters

- **destdir** (`UnionType[str, None]`) – Relative path to ./dist/APPNAME where the libraries should be put.
- **search_patterns** (`list`) – List of dynamic library filename patterns to collect.

`get_module_file_attribute`(*package*)

Get the absolute path to the specified module or package.

Modules and packages *must not* be directly imported in the main process during the analysis. Therefore, to avoid leaking the imports, this function uses an isolated subprocess when it needs to import the module and obtain its `__file__` attribute.

Parameters **package** (*str*) – Fully-qualified name of module or package.

Returns Absolute path of this module.

Return type str

`get_module_attribute`(*module_name*, *attr_name*)

Get the string value of the passed attribute from the passed module if this attribute is defined by this module _or_ raise *AttributeError* otherwise.

Since modules cannot be directly imported during analysis, this function spawns a subprocess importing this module and returning the string value of this attribute in this module.

Parameters

- **module_name** (*str*) – Fully-qualified name of this module.
- **attr_name** (*str*) – Name of the attribute in this module to be retrieved.

Returns String value of this attribute.

Return type str

Raises AttributeError – If this attribute is undefined.

`get_package_paths`(*package*)

Given a package, return the path to packages stored on this machine and also returns the path to this particular package. For example, if pkg.subpkg lives in /abs/path/to/python/libs, then this function returns `(/abs/path/to/python/libs, /abs/path/to/python/libs/pkg/subpkg)`.

NOTE: due to backwards compatibility, this function returns only one package path along with its base directory. In case of PEP 420 namespace package with multiple location, only first location is returned. To obtain all package paths, use the `get_all_package_paths` function and obtain corresponding base directories using the `package_base_path` helper.

`copy_metadata`(*package_name*, *recursive=False*)

Collect distribution metadata so that `importlib.metadata.distribution()` or `pkg_resources.get_distribution()` can find it.

This function returns a list to be assigned to the `datas` global variable. This list instructs PyInstaller to copy the metadata for the given package to the frozen application's data directory.

Parameters

- **package_name** (*str*) – Specifies the name of the package for which metadata should be copied.
- **recursive** (*bool*) – If true, collect metadata for the package's dependencies too. This enables use of `importlib.metadata.requires('package')` or `pkg_resources.require('package')` inside the frozen application.

Returns This should be assigned to `datas`.

Return type list

Examples

```
>>> from PyInstaller.utils.hooks import copy_metadata
>>> copy_metadata('sphinx')
[('c:\python27\lib\site-packages\Sphinx-1.3.2.dist-info',
  'Sphinx-1.3.2.dist-info')]
```

Some packages rely on metadata files accessed through the `importlib.metadata` (or the now-deprecated `pkg_resources`) module. PyInstaller does not collect these metadata files by default. If a package fails without the metadata (either its own, or of another package that it depends on), you can use this function in a hook to collect the corresponding metadata files into the frozen application. The tuples in the returned list contain two strings. The first is the full path to the package's metadata directory on the system. The second is the destination name, which typically corresponds to the basename of the metadata directory. Adding these tuples the the `datas` hook global variable, the metadata is collected into top-level application directory (where it is usually searched for).

Changed in version 4.3.1: Prevent `dist-info` metadata folders being renamed to `egg-info` which broke `pkg_resources.require` with *extras* (see #3033).

Changed in version 4.4.0: Add the **recursive** option.

collect_entry_point(*name*)
Collect modules and metadata for all exporters of a given entry point.

Parameters **name** (str) – The name of the entry point. Check the documentation for the library that uses the entry point to find its name.

Returns A `(datas, hiddenimports)` pair that should be assigned to the `datas` and `hiddenimports`, respectively.

For libraries, such as `pytest` or `keyring`, that rely on plugins to extend their behaviour.

Examples

Pytest uses an entry point called `'pytest11'` for its extensions. To collect all those extensions use:

```
datas, hiddenimports = collect_entry_point("pytest11")
```

These values may be used in a hook or added to the `datas` and `hiddenimports` arguments in the `.spec` file. See *Using Spec Files*.

New in version 4.3.

get_homebrew_path(*formula=''*)
Return the homebrew path to the requested formula, or the global prefix when called with no argument.

Returns the path as a string or None if not found.

include_or_exclude_file(*filename*, *include_list=None*, *exclude_list=None*)
Generic inclusion/exclusion decision function based on filename and list of include and exclude patterns.

Parameters

- **filename** (str) – Filename considered for inclusion.
- **include_list** (UnionType[list, None]) – List of inclusion file patterns.
- **exclude_list** (UnionType[list, None]) – List of exclusion file patterns.

Returns A boolean indicating whether the file should be included or not.

If `include_list` is provided, True is returned only if the filename matches one of include patterns (and does not match any patterns in `exclude_list`, if provided). If `include_list` is not provided, True is returned if filename does not match any patterns in `exclude list`, if provided. If neither list is provided, True is returned for any filename.

collect_delvewheel_libs_directory(*package_name*, *libdir_name=None*, *datas=None*, *binaries=None*)
Collect data files and binaries from the .libs directory of a delvewheel-enabled python wheel. Such wheels ship their shared libraries in a .libs directory that is located next to the package directory, and therefore falls outside the purview of the collect_dynamic_libs() utility function.

Parameters

- **package_name** – Name of the package (e.g., scipy).
- **libdir_name** – Optional name of the .libs directory (e.g., scipy.libs). If not provided, ".libs" is added to `package_name`.
- **datas** – Optional list of datas to which collected data file entries are added. The combined result is retuned as part of the output tuple.
- **binaries** – Optional list of binaries to which collected binaries entries are added. The combined result is retuned as part of the output tuple.

Returns A `(datas, binaries)` pair that should be assigned to the `datas` and `binaries`, respectively.

Return type tuple

Examples

Collect the `scipy.libs` delvewheel directory belonging to the Windows `scipy` wheel:

```
datas, binaries = collect_delvewheel_libs_directory("scipy")
```

When the collected entries should be added to existing `datas` and `binaries` listst, the following form can be used to avoid using intermediate temporary variables and merging those into existing lists:

```
datas, binaries = collect_delvewheel_libs_directory("scipy", datas=datas,␣
↪binaries=binaries)
```

New in version 5.6.

Support for Conda

Additional helper methods for working specifically with Anaconda distributions are found at *PyInstaller.utils.hooks.conda_support* which is designed to mimic (albeit loosely) the importlib.metadata package. These functions find and parse the distribution metadata from json files located in the `conda-meta` directory.

New in version 4.2.0.

This module is available only if run inside a Conda environment. Usage of this module should therefore be wrapped in a conditional clause:

```
from PyInstaller.compat import is_pure_conda

if is_pure_conda:
    from PyInstaller.utils.hooks import conda_support

    # Code goes here. e.g.
    binaries = conda_support.collect_dynamic_libs("numpy")
    ...
```

Packages are all referenced by the *distribution name* you use to install it, rather than the *package name* you import it with. I.e., use `distribution("pillow")` instead of `distribution("PIL")` or use `package_distribution("PIL")`.

distribution(*name*)

Get distribution information for a given distribution **name** (i.e., something you would `conda install`).

Return type *Distribution*

package_distribution(*name*)

Get distribution information for a **package** (i.e., something you would import).

Return type *Distribution*

For example, the package `pkg_resources` belongs to the distribution `setuptools`, which contains three packages.

```
>>> package_distribution("pkg_resources")
Distribution(name="setuptools",
             packages=['easy_install', 'pkg_resources', 'setuptools'])
```

files(*name*, *dependencies=False*, *excludes=None*)
List all files belonging to a distribution.

Parameters

- **name** (str) – The name of the distribution.
- **dependencies** (bool) – Recursively collect files of dependencies too.
- **excludes** (UnionType[list, None]) – Distributions to ignore if **dependencies** is true.

Return type List[*PackagePath*]

Returns All filenames belonging to the given distribution.

With `dependencies=False`, this is just a shortcut for:

```
conda_support.distribution(name).files
```

requires(*name*, *strip_versions=False*)
List requirements of a distribution.

Parameters

- **name** (str) – The name of the distribution.
- **strip_versions** (bool) – List only their names, not their version constraints.

Return type List[str]

Returns A list of distribution names.

class Distribution(*json_path*)
A bucket class representation of a Conda distribution.

This bucket exports the following attributes:

Variables

- **name** – The distribution's name.
- **version** – Its version.
- **files** – All filenames as *PackagePath()*s included with this distribution.
- **dependencies** – Names of other distributions that this distribution depends on (with version constraints removed).
- **packages** – Names of importable packages included in this distribution.

This class is not intended to be constructed directly by users. Rather use *distribution()* or *package_distribution()* to provide one for you.

class PackagePath(**args*, ***kwargs*)
A filename relative to Conda's root (`sys.prefix`).

This class inherits from `pathlib.PurePosixPath` even on non-Posix OSs. To convert to a `pathlib.Path` pointing to the real file, use the *locate()* method.

locate()
Return a path-like object for this path pointing to the file's true location.

walk_dependency_tree(*initial*, *excludes=None*)
Collect a *Distribution* and all direct and indirect dependencies of that distribution.

Parameters

- **initial** (str) – Distribution name to collect from.
- **excludes** (Optional[Iterable[str], None]) – Distributions to exclude.

Returns A {name: distribution} mapping where distribution is the output of *conda_support.distribution(name)*.

collect_dynamic_libs(*name*, *dest='.'*, *dependencies=True*, *excludes=None*)
Collect DLLs for distribution **name**.

Parameters

- **name** (str) – The distribution's project-name.
- **dest** (str) – Target destination, defaults to '.'.
- **dependencies** (bool) – Recursively collect libs for dependent distributions (recommended).
- **excludes** (Optional[Iterable[str], None]) – Dependent distributions to skip, defaults to None.

Returns List of DLLs in PyInstaller's (source, dest) format.

This collects libraries only from Conda's shared lib (Unix) or Library/bin (Windows) folders. To collect from inside a distribution's installation use the regular *PyInstaller.utils.hooks.collect_dynamic_libs()*.

2.13.6 Subprocess isolation with PyInstaller.isolated

PyInstaller hooks typically will need to import the package which they are written for but doing so may manipulate globals such as sys.path or os.environ in ways that affect the build. For example, on Windows, Qt's binaries are added to then loaded via PATH in such a way that if you import multiple Qt variants in one session then there is no guarantee which variant's binaries each variant will get!

To get around this, PyInstaller does any such tasks in an isolated Python subprocess and ships a *PyInstaller.isolated* submodule to do so in hooks.

```
from PyInstaller import isolated
```

This submodule provides:

- *isolated.call()* to evaluate functions in isolation.
- *@isolated.decorate* to mark a function as always called in isolation.
- *isolated.Python()* to efficiently call many functions in a single child instance of Python.

call(*function*, **args*, ***kwargs*)
Call a function with arguments in a separate child Python. Retrieve its return value.

Parameters

- **function** – The function to send and invoke.
- ***args** –
- ****kwargs** – Positional and keyword arguments to send to the function. These must be simple builtin types - not custom classes.

Returns The return value of the function. Again, these must be basic types serialisable by `marshal.dumps()`.

Raises `RuntimeError` – Any exception which happens inside an isolated process is caught and reraised in the parent process.

To use, define a function which returns the information you're looking for. Any imports it requires must happen in the body of the function. For example, to safely check the output of `matplotlib.get_data_path()` use:

```
# Define a function to be ran in isolation.
def get_matplotlib_data_path():
    import matplotlib
    return matplotlib.get_data_path()

# Call it with isolated.call().
get_matplotlib_data_path = isolated.call(matplotlib_data_path)
```

For single use functions taking no arguments like the above you can abuse the decorator syntax slightly to define and execute a function in one go.

```
>>> @isolated.call
... def matplotlib_data_dir():
...     import matplotlib
...     return matplotlib.get_data_path()
>>> matplotlib_data_dir
'/home/brenainn/.pyenv/versions/3.9.6/lib/python3.9/site-packages/matplotlib/mpl-
↪data'
```

Functions may take positional and keyword arguments and return most generic Python data types.

```
>>> def echo_parameters(*args, **kwargs):
...     return args, kwargs
>>> isolated.call(echo_parameters, 1, 2, 3)
(1, 2, 3), {}
>>> isolated.call(echo_parameters, foo=["bar"])
(), {'foo': ['bar']}
```

Notes

To make a function behave differently if it's isolated, check for the `__isolated__` global.

```
if globals().get("__isolated__", False):
    # We're inside a child process.
    ...
else:
    # This is the master process.
    ...
```

`decorate`(*function*)

Decorate a function so that it is always called in an isolated subprocess.

Examples

To use, write a function then prepend `@isolated.decorate`.

```
@isolated.decorate
def add_1(x):
    "Add 1 to ``x``, displaying the current process ID."
    import os
    print(f"Process {os.getpid()}: Adding 1 to {x}.")
    return x + 1
```

The resultant `add_1()` function can now be called as you would a normal function and it'll automatically use a subprocess.

```
>>> add_1(4)
Process 4920: Adding 1 to 4.
5
>>> add_1(13.2)
Process 4928: Adding 1 to 13.2.
14.2
```

class Python(*strict_mode=None*)

Start and connect to a separate Python subprocess.

This is the lowest level of public API provided by this module. The advantage of using this class directly is that it allows multiple functions to be evaluated in a single subprocess, making it faster than multiple calls to *call()*.

The `strict_mode` argument controls behavior when the child process fails to shut down; if strict mode is enabled, an error is raised, otherwise only warning is logged. If the value of `strict_mode` is `None`, the value of `PyInstaller.compat.strict_collect_mode` is used (which in turn is controlled by the `PYINSTALLER_STRICT_COLLECT_MODE` environment variable.

Examples

To call some predefined functions `x = foo()`, `y = bar("numpy")` and `z = bazz(some_flag=True)` all using the same isolated subprocess use:

```
with isolated.Python() as child:
    x = child.call(foo)
    y = child.call(bar, "numpy")
    z = child.call(bazz, some_flag=True)
```

call(*function*, **args*, ***kwargs*)

Call a function in the child Python. Retrieve its return value. Usage of this method is identical to that of the *call()* function.

2.13.7 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.

co: Code object compiled from the contents of __file__ (e.g., via the compile() builtin).

analysis: The Analysis object that loads the hook.

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.

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.

set_module_collection_mode (name, mode): Set the *package collection mode* for the specified package/module name. Valid values for mode are: 'pyz', 'pyc', 'py', 'pyz+py', 'py+pyz' and None. None clears/resets the setting for the given package/module name - but only within the current hook's context! The collection mode may be set for the hooked package, its sub-module or sub-package, or for other packages. If name is None, it is substituted with the hooked package/module name.

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.

Hooks may access the user parameters, given in the hooksconfig argument in the spec file, by calling *get_hook_config()* inside a *hook()* function.

get_hook_config(*hook_api*, *module_name*, *key*)
Get user settings for hooks.

Parameters

- **module_name** (str) – The module/package for which the key setting belong to.
- **key** (str) – A key for the config.

Returns The value for the config. None if not set.

The get_hook_config function will lookup settings in the Analysis.hooksconfig dict.

The hook settings can be added to .spec file in the form of:

```
a = Analysis(["my-app.py"],
    ...
    hooksconfig = {
```

(continues on next page)

(continued from previous page)

```
        "gi": {
            "icons": ["Adwaita"],
            "themes": ["Adwaita"],
            "languages": ["en_GB", "zh_CN"],
        },
    },
    ...
)
```

2.13.8 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.

2.13.9 The `pre_safe_import_module( psim_api )` Method

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

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

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

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

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

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

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

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

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

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

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

The `psim_api` object also offers the following methods:

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

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

`add_alias_module( real_module_name, alias_module_name )`: `real_module_name` is the fully-qualifed 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.

2.14 Hook Configuration Options

As of version 4.4, PyInstaller implements a mechanism for passing configuration options to the hooks. At the time of writing, this feature is supported only in *.spec files* and has no command-line interface equivalent.

The hook configuration options consist of a dictionary that is passed to the `Analysis` object via the `hooksconfig` argument. The keys of the dictionary represent *hook identifiers* while the values are dictionaries of hook-specific keys and values that correspond to hook settings:

```
a = Analysis(
    ["program.py"],
    ...,
    hooksconfig={
        "some_hook_id": {
            "foo": ["entry1", "entry2"],
            "bar": 42,
            "enable_x": True,
        },
        "another_hook_id": {
            "baz": "value",
        },
    },
    ...,
)
```

2.14.1 Supported hooks and options

This section lists hooks that implement support for configuration options. For each hook (or group of hooks), we provide the *hook identifier* and the list of supported options.

GObject introspection (gi) hooks

The options passed under *gi* hook identifier control the collection of GLib/Gtk resources (themes, icons, translations) in various hooks related to `GObject introspection` (i.e., `hook-gi.*`).

They are especially useful when freezing `Gtk3`-based applications on linux, as they allow one to limit the amount of themes and icons collected from the system `/usr/share` directory.

Hook identifier: `gi`

Options

- `languages` [*list of strings*]: list of locales (e.g., ˙en_US˙) for which translations should be collected. By default, `gi` hooks collect all available translations.
- `icons` [*list of strings*]: list of icon themes (e.g., *Adwaita*) that should be collected. By default, `gi` hooks collect all available icon themes.
- `themes` [*list of strings*]: list of Gtk themes (e.g., *Adwaita*) that should be collected. By default, `gi` hooks collect all available icon themes.
- `module-versions` [*dict of version strings*]: versions of gi modules to use. For example, a key of 'GtkSource' and value to '4' will use gtksourceview4.

Example

Collect only `Adwaita` theme and icons, limit the collected translations to British English and Simplified Chinese, and use version 3.0 of Gtk and version 4 of GtkSource:

```
a = Analysis(
    ["my-gtk-app.py"],
    ...,
    hooksconfig={
        "gi": {
            "icons": ["Adwaita"],
            "themes": ["Adwaita"],
            "languages": ["en_GB", "zh_CN"],
            "module-versions": {
                "Gtk": "3.0",
                "GtkSource": "4",
            },
        },
    },
    ...,
)
```

Note: Currently the `module-versions` configuration is available only for `GtkSource`, `Gtk`, and `Gdk`.

GStreamer (gi.repository.Gst) hook

The collection of GStreamer is subject to both the general `gi` hook configuration (for example, collection of translations file as controlled by the `languages` option) and by special hook configuration named `gstreamer`[1] that controls collection of GStreamer plugins.

The GStreamer framework comes with a multitude of plugins that are typically installed as separate packages (`gstreamer-plugins-base`, `gstreamer-plugins-good`, `gstreamer-plugins-bad`, and `gstreamer-plugins-ugly`; the naming varies between packaging systems). By default, PyInstaller collects *all* available plugins as well as their binary dependencies; therefore, having all GStreamer plugins installed in the build environment will likely result in collection of many unnecessary plugins and increased frozen application size due to potential complex dependency chains of individual plugins and the underlying shared libraries.

Hook identifier: `gstreamer`[?]

Options

- `include_plugins` [*list of strings*]: list of plugin names to include in the frozen application. Specifying the include list implicitly excludes all plugins that do not appear in the list.
- `exclude_plugins` [*list of strings*]: list of plugin names to exclude from the frozen application. If include list is also available, the exclude list is applied after it; if not, the exclude list is applied to all available plugins.

Both include and exclude list expect base plugin names (e.g., `audioparsers`, `matroska` , `x264`, `flac`). Internally, each name is converted into a pattern (e.g., `'**/*flac.*'`), and matched using `fnmatch` against actual plugin file names. Therefore, it is also possible to include the wildcard (*) in the plugin name[2].

Basic example: excluding an unwanted plugin

Exclude the `opencv` GStreamer plugin to prevent pulling OpenCV shared libraries into the frozen application.

```
a = Analysis(
    ["my-gstreamer-app.py"],
    ...,
    hooksconfig={
        "gstreamer": {
            "exclude_plugins": [
                "opencv",
            ],
        },
    },
    ...,
)
```

Advanced example: including only specific plugins

When optimizing the frozen application size, it is often more efficient to explicitly include only the subset of the plugins that are actually required for the application to function.

Consider the following simple player application:

```
# audio_player.py
import sys
import os
```

(continues on next page)

[1] While the hook is called `gi.repository.Gst`, the identifier for Gstreamer-related options was chosen to be simply `gstreamer`.

[2] And it is also possible to get away with accidentally specifying the plugin prefix, which is typically *libgst*, but can also be *gst*, depending on the toolchain that was used to build GStreamer.

(continued from previous page)

```
import gi
gi.require_version('Gst', '1.0')
from gi.repository import GLib, Gst

if len(sys.argv) != 2:
    print(f"Usage: {sys.argv[0]} <filename>")
    sys.exit(-1)

filename = os.path.abspath(sys.argv[1])
if not os.path.isfile(filename):
    print(f"Input file {filename} does not exist!")
    sys.exit(-1)

Gst.init(sys.argv)
mainloop = GLib.MainLoop()

playbin = Gst.ElementFactory.make("playbin", "player")
playbin.set_property('uri', Gst.filename_to_uri(filename))
playbin.set_property('volume', 0.2)
playbin.set_state(Gst.State.PLAYING)

mainloop.run()
```

Suppose that, although the application is using the generic `playbin` and `player` elements, we intend for the frozen application to play only audio files. In that case, we can limit the collected plugins as follows:

```
# The not-completely-optimized list of gstreamer plugins for playing a FLAC
# (and possibly some other) audio files on linux and Windows.
gst_include_plugins = [
    # gstreamer
    "coreelements",
    # gstreamer-plugins-base
    "alsa",  # Linux audio output
    "audioconvert",
    "audiomixer",
    "audiorate",
    "audioresample",
    "ogg",
    "playback",
    "rawparse",
    "typefindfunctions",
    "volume",
    "vorbis",
    # gstreamer-plugins-good
    "audioparsers",
    "auparse",
    "autodetect",
    "directsound", # Windows audio output
    "flac",
    "id3demux",
    "lame",
    "mpg123",
```

(continues on next page)

(continued from previous page)

```
    "osxaudio",  # macOS audio output
    "pulseaudio",  # Linux audio output
    "replaygain",
    "speex",
    "taglib",
    "twolame",
    "wavparse",
    # gstreamer-plugins-bad
    "wasapi",  # Windows audio output
]

a = Analysis(
    ["audio_player.py"],
    ...,
    hooksconfig={
        "gstreamer": {
            "include_plugins": gst_include_plugins,
        },
    },
    ...,
)
```

Determining which plugins need to be collected may require good knowledge of GStreamer pipelines and their plugin system, and may result in several test iterations to see if the required multimedia functionality works as expected. Unfortunately, there is no free lunch when it comes to optimizing the size of application that uses a plugin system like that. Keep in mind that in addition to obviously-named plugins (such as `flac` for FLAC-related functionality), you will likely need to collect at least some plugins that come from `gstreamer` itself (e.g., the `coreelements` one) and at least some that are part of `gstreamer-plugins-base`.

Matplotlib hooks

The hooks for the `matplotlib` package allow user to control the backend collection behavior via `backends` option under the `matplotlib` identifier, as described below.

Hook identifier: `matplotlib`

Options

- `backends` [*string* or *list of strings*]: backend selection method or name(s) of backend(s) to collect. Valid string values: `'auto'`, `'all'`, or a human-readable backend name (e.g., `'TkAgg'`). To specify multiple backends to be collected, use a list of strings (e.g., `['TkAgg', 'Qt5Agg']`).

Backend selection process

If `backends` option is set to `'auto'` (or not specified), the hook performs auto-detection of used backends, by scanning the code for `matplotlib.use()` function calls with literal arguments. For example, `matplotlib.use('TkAgg')` being used in the code results in the `TkAgg` backend being collected. If no such calls are found, the default backend is determined as the first importable GUI-based backend, using the same priority list as internally used by the `matplotlib.get_backend()` and `matplotlib.pyplot.switch_backend()` functions: `['MacOSX', 'Qt5Agg', 'Gtk3Agg', 'TkAgg', 'WxAgg']`. If no GUI-based backend is importable, the headless `'Agg'` is collected instead.

Note: Due to limitations of the bytecode-scanning approach, only specific forms of `matplotlib.use()` invocation can be automatically detected. The backend must be specified as string literal (as opposed to being passed via a

variable). The second optional argument, `force`, can also be specified, but it must also be a literal and must not be specified as a keyword argument:

```
import matplotlib

matplotlib.use('TkAgg')  # detected
matplotlib.use('TkAgg', False)  # detected

backend = 'TkAgg'
matplotlib.use(backend)  # not detected

matplotlib.use('TkAgg', force=False)  # not detected
```

In addition to `matplotlib` module name, its common alias, `mpl` is also recognized:

```
import matplotlib as mpl
mpl.use('TkAgg')  # detected
```

Importing the function from the module should also work:

```
from matplotlib import use
use('TkAgg')  # detected
```

If `backends` option is set to `'all'`, all (importable) backends are selected, which corresponds to the behavior of PyInstaller 4.x and earlier. The list of importable backends depends on the packages installed in the environment; for example, the `Qt5Agg` backend becomes importable if either the `PyQt5` or the `PySide2` package is installed.

Otherwise, the value of the `backends` option is treated as a backend name (if it is a string) or a list of backend names (if it is a list). In the case of user-provided backend names, no additional validation is performed; the backends are collected regardless of whether they are importable or not.

Example

```
a = Analysis(
    ["my-matplotlib-app.py"],
    ...,
    hooksconfig={
        "matplotlib": {
            "backends": "auto",  # auto-detect; the default behavior
            # "backends": "all",  # collect all backends
            # "backends": "TkAgg",  # collect a specific backend
            # "backends": ["TkAgg", "Qt5Agg"],  # collect multiple backends
        },
    },
    ...,
)
```

Note: The `Qt5Agg` backend code conditionally imports all Qt bindings packages (`PySide2`, `PyQt5`, `PySide6`, and `PyQt6`). Therefore, if all are installed in your environment, PyInstaller will end up collecting all. In addition to increasing the frozen application's size, this might also cause conflicts between the collected versions of the shared libraries. To prevent that, use the *`--exclude-module`* option to exclude the extraneous Qt bindings packags (i.e., if you want to use `PyQt5`, use `--exclude-module PySide2`, `--exclude-module PyQt6`, and `--exclude-module PySide6`).

Starting with PyInstaller 6.5, multiple Qt bindings in a frozen application are explicitly disallowed - the build process

aborts with an error if hooks for more than one Qt bindings package are executed. Therefore, `matplotlib` hook automatically attempts to select Qt bindings to use, based on the following heuristics: first, we check whether hooks for any Qt bindings have already been run; if they have, those bindings are selected. If not, the `QT_API` environment variable is read; if it is set and contains a valid Qt bindings package name, those bindings are selected. If not, one of the available Qt bindings is selected. Once a Qt bindings package is selected, all other (potentially available) Qt bindings packages are excluded from the hooked module, to prevent their collection due to conditional imports in the hooked module.

This means that if your entry-point script explicitly imports a Qt bindings package before importing `matplotlib`, those bindings should be chosen automatically. On the other hand, if your program uses `matplotlib` without importing Qt bindings on its own, the Qt bindings to be collected are auto-selected, based on what is available in the build environment. This auto-selection can be overridden by setting the `QT_API` environment variable before running PyInstaller. In this particular case, an environment variable is used instead of hooks configuration mechanism because Qt bindings selection might be performed across several hooks for different packages.

2.14.2 Adding an option to the hook

Implementing support for hook options requires access to `hook_api` object, which is available only when hook implements the `hook(hook_api)` function (as described *here*).

The value of a hook's configuration option can be obtained using the *`get_hook_config()`* function:

```
# hook-mypackage.py
from PyInstaller.utils.hooks import get_hook_config

# Processing unrelated to hook options, using global hook values
binaries, datas, hiddenimports = ...

# Collect extra data
def hook(hook_api):
    # Boolean option 'collect_extra_data'
    if get_hook_config(hook_api, 'mypackage', 'collect_extra_data'):
        extra_datas = ...  # Collect extra data
        hook_api.add_datas(extra_datas)
```

After implementing option handling in the hook, please add a section documenting it under *Supported hooks and options*, to inform the users of the option's availability and the meaning of its value(s).

The above hook example allows the user to toggle the collection of extra data from `mypackage` by setting the corresponding option in their *.spec file*:

```
a = Analysis(
    ["program-using-mypackage.py"],
    ...,
    hooksconfig={
        "mypackage": {
            "collect_extra_data": True,
        },
    },
    ...,
)
```

2.15 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 **`waf`**,
- *git clone* or download the source from our GitHub repository,
- `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` (macOS and Windows only), and
- `../PyInstaller/bootloader/`*`OS_ARCH`*`/runw_d` (macOS 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:

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

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

By setting the environment variable `PYINSTALLER_COMPILE_BOOTLOADER` the pip setup will attempt to build the bootloader for your platform, even if it is already present. Doing so would execute the command `python ./waf configure all` upon installation. You can also pass additional arguments to the build process by setting the `PYINSTALLER_BOOTLOADER_WAF_ARGS` environment variable.

Supported platforms are

- GNU/Linux (using gcc)
- Windows (using Visual C++ (VS2015 or later) or MinGW's gcc)
- Mac OX X (using clang)

Contributed platforms are

- AIX (using gcc or xlc)
- HP-UX (using gcc or xlc)
- Solaris

For more information about cross-building please read on and mind the section about the virtual machines provided in the Vagrantfile.

2.15.1 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 zlib1g-dev
```

- On Fedora, RedHat and derivates:

```
sudo yum groupinstall "Development Tools"
sudo yum install zlib-devel
```

- 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).

Cross Building for Different Architectures

Bootloaders can be built for other architectures such as ARM or MIPS using Docker. The Dockerfile contains the instructions on how to do this. Open it in some flavour of text previewer to see them:

```
less bootloader/Dockerfile
```

2.15.2 Building for macOS

On macOS please install Xcode, Apple's suite of tools for developing software for macOS. Instead of installing the full *Xcode* package, you can also install and use Command Line Tools for Xcode. Installing either will provide the *clang* compiler.

If the toolchain supports *universal2* binaries, the 64-bit bootloaders are by default built as *universal2* fat binaries that support both *x86_64* and *arm64* architectures. This requires a recent version of *Xcode* (12.2 or later). On older toolchains that lack support for *universal2* binaries, a single-arch *x86_64* thin bootloader is built. This behavior can be controlled by passing `--universal2` or `--no-universal2` flags to the `waf` build command. Attempting to use `--universal2` flag and a toolchain that lacks support for *universal2* binaries will result in configuration error.

The `--no-universal2` flag leaves the target architecture unspecified letting the resultant executable's architecture be the C compiler's default (which is almost certainly the architecture of the build machine). Should you want to build a thin executable of either architecture, use the `--no-universal2` flag and then optionally override the compiler, adding the `-arch` flag, via the `CC` environment variable.

Build a thin, native executable:

```
python waf --no-universal2 all
```

Build a thin, `x86_64` executable (regardless of the build machine's architecture):

```
CC='clang -arch x86_64' python waf --no-universal2  all
```

Build a thin, `arm64` executable (regardless of the build machine's architecture):

```
CC='clang -arch arm64' python waf --no-universal2 all
```

By default, the build script targets macOS 10.13, which can be overridden by exporting the MACOSX_DEPLOYMENT_TARGET environment variable.

Cross-Building for macOS

For cross-compiling for macOS 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 macOS SDK.

This is easy to get and needs to be done only once and the result can be transferred to you build-system. The build-system can then be a normal (somewhat current) GNU/Linux system.[1]

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 toolchain. If you are interested in the details, and what other features OS X Cross offers, please refer to its homepage.

To save you reading the OSXCross' documentation, we prepared a virtual box definition that performs all required steps. If you are interested in the precise commands, please refer to `packages_osxcross_debianoid`, `prepare_osxcross_debianiod`, and `build_osxcross` in the Vagrantfile.

Please proceed as follows:

1. Download Command Line Tools for Xcode 12.2 or later. You will need an *Apple ID* to search and download the files; if you do not have one already, you can register it for free.

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

2. Save the downloaded *.dmg* file to `bootloader/_sdks/osx/Xcode_tools.dmg`.

3. 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`.

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

[1] 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.

Building the Bootloader

Again, simply use the Vagrantfile to automatically build the macOS bootloaders:

```
export TARGET=OSX  # make the Vagrantfile build for macOS
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:

   ```
   vagrant destroy build-osxcross
   ```

4. If you are finished with the macOS bootloaders, unset *TARGET* again:

   ```
   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):

```
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
```

2.15.3 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 build using Visual Studio C++ compiler.

   Visual Studio 2015 or later is required.

2. Using the MinGW-w64 suite.

   This allows to create smaller, dynamically linked executables, but requires to use the same level of Visual Studio[2] 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 is 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`,

[2] 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.

- 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.
- 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:

```
... one-line-install as written on the chocolatey homepage
choco install -y python3 visualstudio2019-workload-vctools
```

- Useful Links:
  - Microsoft Visual C++ Build-Tools 2015
  - Microsoft Build-Tools for Visual Studio 2017.

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:

```
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.

2.15.4 Building for AIX

- By default AIX builds 32-bit executables.
- For 64-bit executables set the environment variable `OBJECT_MODE`.

If Python was built as a 64-bit executable then the AIX utilities that work with binary files (e.g., .o, and .a) may need the flag `-X64`. Rather than provide this flag with every command, the preferred way to provide this setting is to use the environment variable `OBJECT_MODE`. Depending on whether Python was build as a 32-bit or a 64-bit executable you may need to set or unset the environment variable `OBJECT_MODE`.

To determine the size the following command can be used:

```
$ python -c "import sys; print(sys.maxsize <= 2**32)"
True
```

When the answer is `True` (as above) Python was build as a 32-bit executable.

When working with a 32-bit Python executable proceed as follows:

```
unset OBJECT_MODE
./waf configure all
```

When working with a 64-bit Python executable proceed as follows:

```
export OBJECT_MODE=64
./waf configure all
```

Note: The correct setting of `OBJECT_MODE` is also needed when you use PyInstaller to package your application.

To build the bootloader you will need a compiler compatible (identical) with the one used to build python.

Note: Python compiled with a different version of gcc that you are using might not be compatible enough. GNU tools are not always binary compatible.

If you do not know which compiler that was, this command can help you determine if the compiler was gcc or an IBM compiler:

```
python -c "import sysconfig; print(sysconfig.get_config_var('CC'))"
```

If the compiler is gcc you may need additional RPMs installed to support the GNU run-time dependencies.

When the IBM compiler is used no additional prerequisites are expected. The recommended value for `CC` with the IBM compilers is *:command:xlc_r*.

2.15.5 Building for FreeBSD

A FreeBSD bootloader may be built with clang using *the usual steps* on a FreeBSD machine. Beware, however that any executable compiled natively on FreeBSD will only run on equal or newer versions of FreeBSD. In order to support older versions of FreeBSD, you must compile the oldest OS version you wish to support.

Alternatively, the FreeBSD bootloaders may be cross compiled from Linux using Docker and a FreeBSD cross compiler image. This image is kept in sync with the oldest non end of life FreeBSD release so that anything compiled on it will work on all active FreeBSD versions.

In a random directory:

- Start the docker daemon (usually with `systemctl start docker` - possibly requiring `sudo` if you haven't setup rootless docker).
- Download the latest cross compiler `.tar.xz` image from here.
- Import the image: `docker image load -i freebsd-cross-build.tar.xz`. The cross compiler image is now saved under the name `freebsd-cross-build`. You may discard the `.tar.xz` file if you wish.

Then from the root of this repository:

- Run:

```
docker run -v $(pwd):/io -it freebsd-cross-build bash -c "cd /io/bootloader; ./waf␣
↪all"
```

2.15.6 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[3] will automatically build the bootloader when running *vagrant up GUEST* or *vagrant provision GUEST*. They will build both 32- and 64-bit bootloaders.

When building the bootloaders, the guests 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 linux64
```

or like this:

[3] Except of guest *osxcross*, which will build the OS X SDK and cctools as described in section *Cross-Building for macOS*.

```
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=OSX` is set, cross-builds the bootloaders for macOS (see *Cross-Building for macOS*).
- 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.

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 macOS*.

2.16 Changelog for PyInstaller

2.16.1 6.6.0 (2024-04-13)

Features

- (Windows) Implement support for resolving executable's true location when launched via a symbolic link. (#8300)
- Implement an option to explicitly specify the bytecode optimization level for collected python code, independent of the optimization level in the python process under which PyInstaller is running. At the .spec file level, this is controlled by optional `optimize` argument in the `Analysis` constructor. At the CLI level, this is controlled by new `--optimize` command-line option, which sets the `optimize` argument for `Analysis` as well as *interpreter run-time options* in the generated spec file. See *Bytecode Optimization Level* for details. (#8252)

Bugfix

- (macOS) Explicitly convert the value of `version` argument to `BUNDLE` into a string, in order to mitigate cases when user accidentally enters an integer or a float. The version value ends up being written to `Info.plist` as the `CFBundleShortVersionString` entry, and if this entry is not of a string type (for example, is an integer), the generated .app bundle crashes at start. (#4466)
- (Windows) Avoid trying to import `PySimpleGUI` in the subprocess that analyzes dynamic library search modifications made by packages prior to the binary dependency analysis. When imported for the first time, `PySimpleGUI` 5.x displays a "first-run" dialog, which poses a problem for unattended PyInstaller builds running in a clean environment, for example, in a CI pipeline. (#8396)

- (Windows) Implement a work-around for running PyInstaller under python process with `-OO` (or `PYTHONOPTIMIZE=2`) with `cffi` installed. We now temporarily disable import of `cffi` while importing `pywin32-ctypes` in `PyInstaller.compat` to ensure that `ctypes` backend is always used, as the `cffi` backend uses `pycparser` and requires docstrings, which makes it incompatible with the `-OO` mode. (#6345)

Hooks

- Update `PySide6.Qt3DRender` hook for compatibility with `PySide6` 6.7.0 (add hidden import for `PySide6.QtOpenGL` module). (#8404)
- Update `scipy.special._ufuncs` hook for compatibility with SciPy 1.13.0 (add `scipy.special._cdflib` to hidden imports). (#8394)

Bootloader

- (Windows) Attempt to shorten the duration of spinning-wheel cursor when launching applications built in `windowed` / `noconsole` mode. (#8359)

Documentation

- Add a new documentation section, *Bytecode Optimization Level*, which the describes the new canonical way to control bytecode optimization level of the collected python code. (#8252)
- Add a note to *Specifying Python Interpreter Options* to inform user that setting the optimization level to the application's embedded python interpreter by itself does not result in bytecode optimization of modules that have been collected in byte-compiled form (i.e., the majority of them). (#8252)

2.16.2 6.5.0 (2024-03-09)

Features

- (Linux) Extend the mechanism for collection of `.hmac` files from #8288 to `.chk` files that are used by NSS libraries. (#8315)

Bugfix

- (Linux) Fix collection of `QtWebEngineProcess` helper when collecting Qt (and `PySide`/`PyQt` bindings) installed via Linux distribution packages. In such scenarios, we now force collection of the helper executable into `libexec` directory inside the Qt sub-directory of the bindings' package directory, in order to match the PyPI wheel layout. (#8315)
- (Linux) Fix regression that caused `locale.getlocale()` in frozen applications created with PyInstaller v6.x to return `(None, None)` instead of user-preferred locale. (#8306)
- (Windows) Avoid trying to import `pyqtgraph.canvas` in the subprocess that analyzes dynamic library search modifications made by packages prior to the binary dependency analysis. Trying to import `pyqtgraph.canvas` causes python interpreter to crash under certain circumstances (the issue is present in `pyqtgraph` <= 0.13.3). (#8322)
- (Windows) Fix collection of `QtWebEngineProcess` helper when collecting `PySide2` and Qt installed via Anaconda on Windows. The helper executable is now collected into top-level `PySide2` package directory, in order to match the PyPI wheel layout. (#8315)

- (Windows) Suppress warnings about unresolvable UCRT DLLs (`api-ms-win-*.dll`) on Windows 11. (#8339)
- Fix bootloaders not being found when running an Intel build of Python on Windows ARM64. (#8219)

Incompatible Changes

- PyInstaller now explicitly disallows attempts to collect multiple Qt bindings packages (`PySide2`, `PySide6`, `PyQt5`, `PyQt6`) into a frozen application. When hooks for more than one top-level Qt bindings package are executed, the build process is aborted with error message. This restriction applies across all instances of `Analysis` within a single build (i.e., a single .spec file).

  If you encounter build errors caused by this new restriction, either clean up your build environment (remove the bindings that you are not using), or explicitly exclude the extraneous bindings using *`--exclude-module`* (or equivalent `excludes` list passed as argument to `Analysis` in the .spec file).

  The automatic exclusion of extraneous bindings needs to be done via hooks on per-package basis, so please report problematic packages so that we can write hooks for them. (#8329)

Hooks

- (Linux) When searching for dynamically-loaded NSS libraries during collection of `QtWebEnginge`, account for the possibility of said libraries being either in a separate `nss` directory or in the main library directory. This fixes problems with missing NSS libraries on contemporary Linux distributions that do not use separate `nss` directory (anymore). (#8315)
- Add a hook for `pandas.io.clipboard` to exclude the conditional import of `PyQt5` from this module; the module primarily uses `qtpy` as its Qt bindings abstraction, and the conditional import of `PyQt5` interferes with Qt bindings selection done by our `qtpy` hook. (#8329)
- Add hook for `qtpy` to prevent collection of multiple available Qt bindings. The hook attempts to select a single Qt bindings package and exclude all other Qt bindings packages with the help of the `PyInstaller.utils.hooks.qt.exclude_extraneous_qt_bindings` helper. (#8329)
- Extend hooks for `matplotlib` to prevent collection of multiple available Qt bindings. The new hook for `matplotlib.backends.qt_compat` attempts to select a single Qt bindings package via the following logic implemented in the `PyInstaller.utils.hooks.qt.exclude_extraneous_qt_bindings` helper: first, we check if hooks for any Qt bindings package have already been run; if they had, those bindings are selected. If not, we check for user-specified bindings in the `QT_API` environment variable; if valid bindings name is specified, those bindings are selected. Otherwise, we select one of available bindings. Once a Qt bindings package is selected, the imports of all other Qt bindings packages are excluded from the hooked package. (#8329)
- Have run-time hooks for Qt bindings (`PySide2`, `PySide6`, `PyQt5`, and `PyQt6`) check for presence of the embedded `:/qt/etc/qt.conf` resource, and if not present, inject their own version. This aims to ensure that the bundled Qt is always relocatable, even if the package does not perform injection of embedded `qt.conf` file (most notably, this seems to be the case with `PySide2` collected from Linux distribution packages, and `PySide2` collected from Anaconda on Windows, Linux, and macOS). (#8315)
- PyInstaller now explicitly disallows attempts to collect multiple Qt bindings packages (`PySide2`, `PySide6`, `PyQt5`, `PyQt6`) into a frozen application. When hooks for more than one top-level Qt bindings package are executed, the build process is aborted with error message that informs user of the situation and what to do about it (i.e., exclusion of extraneous packages). The limitation applies to all analyses within a spec file. (#8329)
- Remove run-time hook for `win32com`, as per discussion in issue:*8309*. (#8313)
- Update hook for `matplotlib.backends` to include `QtAgg` and `Gtk4Agg` in the list of backend candidates. (#8334)

Bootloader

- Have bootloader set the `configure_locale` field in the interpreter pre-config structure, so that user-preferred locale is set during interpreter pre-initialization. (#8306)

Bootloader build

- The target architecture on Windows using MSVC now defaults to that of the current Python environment – not the current OS. (#8219)

2.16.3 6.4.0 (2024-02-10)

Features

- (Linux) Collect `.hmac` files accompanying shared libraries, if such files are available. This allows frozen application to run on FIPS-enabled Red Hat Enterprise systems, where HMAC is required by self-check implemented by the OpenSSL crypto library. Furthermore, ensure that shared libraries with accompanying `.hmac` files are exempted from any additional processing (for example, when building with `--strip` option) to avoid invalidating the HMAC. (#8273)
- (Windows) Make bootloader codepaths involved in creation of temporary directories for `onefile` builds AppContainer-aware. If the process runs inside an AppContainer, the temporary directory's DACL needs to explicitly include the AppContainerSID, otherwise the directory becomes inaccessible to the process. (#8291)
- (Windows) Make Windows implementation of PyInstaller's `_pyi_rth_utils.tempdir.secure_mkdir` (used by `matplotlib` and `win32com` run-time hooks to create temporary directories) AppContainer-aware. If the process runs inside an AppContainer, the temporary directory's DACL needs to explicitly include the AppContainerSID, otherwise the directory becomes inaccessible to the process. (#8290)
- Implement strict Qt dependency validation for collection of Qt plugins and QML components/plugins. We now perform preliminary binary dependency analysis of the plugins, and automatically exclude plugins that have at least one missing Qt dependency. This prevents collection of plugins that cannot be used anyway because of a missing Qt shared library (that is, for example, omitted from a PyPI wheel). Furthermore, we disallow Qt dependencies of a plugin to be resolved outside of the primary location of Qt shared libraries, in order to prevent missing dependencies from pulling in Qt libraries from alternative locations that happen to be in the search path (for example, when using `PyQt5` PyPI wheels while also having a system-installed Qt5 on Linux, a Homebrew-installed Qt5 on macOS, or a custom Windows Qt5 build that happens to be in `PATH`). (#8226)

Bugfix

- (Linux) Prevent collection of `libcuda.so.1`, which is part of NVIDIA driver and must match the rest of the driver's components. Collecting a copy might lead to issues when build and target system use different versions of NVIDIA driver. (#8278)
- (macOS) When validating the macOS SDK version of collected binaries, handle errors raised by `osxutils.get_macos_sdk_version`; log a warning about failed version query, and add the offending binary to the list of potentially problematic binaries to warn the user about. (#8220)
- Fix `pkgutil.iter_modules` override to gracefully handle cases when the given path corresponds to a module instead of a package. (#8191)
- Prevent Qt and QML plugins with missing Qt dependencies in the `PySide2`, `PyQt5`, `PySide6`, and `PyQt6` PyPI wheels from pulling in Qt shared libraries from alternative locations (for example, system-installed Qt on Linux, Homebrew-installed Qt on macOS, or a custom Windows Qt build that happens to be in `PATH`), and resulting in a frozen application that contains an incompatible mix of Qt libraries. (#8087)

- Switch the hashing function in PyInstaller's binary cache from MD5 to SHA1, as the former cannot be used on FIPS-enabled Red Hat Enterprise Linux systems. (#8288)
- When trying to run `pyinstaller` (or equivalent `python -m PyInstaller`) against non-existing script file(s), exit immediately - without trying to write the .spec file and building it. This prevents us from overwriting an existing (and customized) .spec file if user makes a typo in the .spec file's suffix when trying to build it, for example, `pyinstaller program.cpes`. (#8279)

Hooks

- (macOS) Have `PySide6` and `PyQt6` run-time hooks prepend `sys._MEIPASS` to `DYLD_LIBRARY_PATH` in POSIX builds, in order to ensure that `QtNetwork` discovers the bundled copy of the OpenSSL shared library. (#8226)
- Extend the OpenSSL shared library collection in the `QtNetwork` hook helper for `PySide2`, `PyQt5`, `PySide6`, and `PyQt6` to cover all applicable versions of OpenSSL (1.0.2, 1.1.x, 3.x). In addition to Windows, the OpenSSL shared library is now also collected on Linux and macOS. (#8226)

Bootloader

- (Windows) Update the bundled zlib sources to v1.3.1. (#8292)

Documentation

- Add a new documentation chapter, called *Common Issues and Pitfalls*, to cover topics such as launching external programs from frozen applications, multi-processing via `multiprocessing` (specifically, the requirement to call `multiprocessing.freeze_support()`), use of symbolic links in POSIX builds in PyInstaller >= 6.0 and its implications for distribution (e.g., when copying frozen application, or creating `zip` archives), `sys.stdout` and `sys.stderr` being `None` in Windows no-console builds. (#8214)
- Cleanup docstrings to remove mention of `exec_command_stdout`. (#8173)
- Update the *Building macOS App Bundles* section to reflect the layout of macOS app bundles as produced by PyInstaller 6.0 and later. Add a note to discourage use of onefile .app bundles. (#8214)
- Update the introduction part of the *Understanding PyInstaller Hooks* section. (#8214)

2.16.4 6.3.0 (2023-12-10)

Bugfix

- (Linux) Optimize the automatic binary-vs-data classification by avoiding `objdump` based check on files that do not have ELF signature. This mitigates noticeably longer analysis times for projects with large number of (data) files. (#8148)
- (Windows) Add Windows error code 110 (`ERROR_OPEN_FAILED`) to the list of error codes eligible for the retry mechanism that attempts to mitigate build failures due to anti-virus program interference. (#8138)
- (Windows) Fix issue with non-functional `time.sleep()` when building program with Python <= 3.8.6 or Python 3.9.0. (#8104)
- (Windows) Fix issue with splash screen in `onefile` mode failing to extract `VCRUNTIME140.dll` from the archive due to character-case mismatch. We now perform case-insensitive comparison between the name listed in splash dependency list and the names in archive TOC. (#8103)
- Fix PEP 597 EncodingWarnings when `PYTHONWARNDEFAULTENCODING` is set to true. (#8117)

- Fix pre-safe-import hooks for `six.moves`, `urllib3.packages.six.moves`, and `setuptools.extern.six.moves` to gracefully handle cases when the corresponding `six` package is unavailable, as the hook may end up being executed even in that case. (#8145)
- Fix symbolic link tracking in `MERGE` processing, so that distinct symbolic links with same relative target (e.g. `Current -> A` symbolic links in Qt .framework bundles collected on macOS) are properly processed, and kept in the original TOC upon their first occurrence. (#8124)

Hooks

- Add hook for `gi.repository.DBus`. (#8149)
- Add hooks for `gi.repository.AppIndicator3` and `gi.repository.AyatanaAppIndicator3`. (#8149)

Bootloader

- When setting up embedded Python interpreter configuration, set `PyConfig.install_signal_handlers=1` to install signal handlers. This matches the behavior of PyInstaller 5.x bootloaders, where interpreter was initialized via `Py_Initialize()`, which in turn calls `Py_InitializeEx(1)`, i.e., with `install_sigs=1`. (#8105)

2.16.5 6.2.0 (2023-11-11)

Features

- (macOS) At the end of analysis, verify the macOS SDK version reported by binaries to be collected, and warn when the version is either invalid (0.0.0) or too low (< 10.9.0). Such binaries will likely cause issues with code-signing and hardened runtime. (#8043)
- If the `argcomplete` Python module is installed, PyInstaller will use it enable tab completion for its CLI tools. PyInstaller CLIs can still be used without this optional dependency. To install `argcomplete` with PyInstaller, you can put `pyinstaller[completion]` in your dependencies. See also the argcomplete documentation. (#8008)

Bugfix

- (macOS) Fix the bug in binary processing and caching that would update the binary cache index before performing macOS-specific processing (architecture validation, path rewriting). If, for example, architecture validation failed during a build, subsequent build attempts with enabled binary cache (i.e., without the *`--clean`* option) would pick up the partially-processed binary file from the cache, bypassing the architecture validation. NOTE: the existing binary caches need to be purged manually (using *`--clean`* option once) for the fix to take effect! (#8068)
- (macOS) Prevent collection of `.DS_Store` files, which might be present in build environment's package directories after user navigated them using the Finder app. (#8042)
- (Windows) Fix marshal error at the start of binary dependency analysis, caused by inferred DLL search path ending up an instance of `pathlib.Path` instead of `str`. (#8081)
- Bump the required `packaging` version to 22.0, which is required for proper handling of metadata that contains markers with `extras`. (#8061)
- Fix erroneous DLL parent path preservation when `sys.base_prefix` itself is a symbolic link. In such case, we need to exclude both resolved and unresolved path variant for `sys.base_prefix`, in order to prevent either from ending up in the list of directories for which DLL parent paths are preserved. Failing to do so, for example, caused `_ctypes` failing to load in an application build on Windows with Python installed via `scoop`, due to

`libffi-8.dll` having spuriously preserved the parent directory path instead of being collected to top-level application directory. (#8023)

- Fix matching of pre-release versions in *PyInstaller.utils.hooks.check_requirement()* and *PyInstaller.utils.hooks.is_module_satisfies()*. Both functions now match pre-release versions, which restores the behavior of the old `pkg_resources`-based implementation from PyInstaller < 6.0 that is implicitly expected by existing hooks. (#8093)
- If the entry-point script has no suffix, append the `.py` suffix to the filename passed to the `compile` function when byte-compiling the script for collection. This ensures that the entry-point script filename never coincides with executable filename, especially in POSIX builds, where executables have no suffix either (and their name is based on the entry-point script basename by default). Entry-point script having the same filename as the executable causes issues when `traceback` (and `linecache`) try to access source code for it, an in the process end up reading the executable file if it happens to be in the current working directory. (#8046)
- Improve speed of `pkgutil.iter_modules()` override, especially in cases when the function is called multiple times. (#8058)
- Load PyInstaller hooks using **PEP 451** `importlib.abc.Loader.exec_module` instead of deprecated **PEP 302** `importlib.abc.Loader.load_module`. (#8031)
- Prevent an attempt at relative import of a missing (optional) sub-module within a package (e.g., `from .module import something`) from tricking the modulegraph/analysis into collecting an unrelated but eponymous top-level module. (#8010)

Hooks

- Add hook for `PySide6.QtGraphs` that was introduced in `PySide6` 6.6.0. (#8021)
- Add hooks for `distutils.command.check` and `setuptools._distutils.command.check` that prevent unnecessary collection of `docutils` (which in turn triggers collection of `pygments`, `PIL`, etc.). (#8053)
- Deduplicate and sort the list of discovered/selected `matplotlib` backends before displaying it in log messages, to avoid giving impression that they are collected multiple times. (#8009)
- Update `PySide6` hooks for compatibility with `PySide6` 6.6.0 and python 3.12. (#8021)

2.16.6 6.1.0 (2023-10-13)

Features

- Allow users to re-enable the old onedir layout (without contents directory) by settings the *--contents-directory* option (or the equivalent `contents_directory` argument to `EXE` in the .spec file) to `'.'`. (#7968)

Bugfix

- (macOS) Prevent bootloader from clearing `DYLD_*` environment variables when running in `onefile` mode, in order to make behavior consistent with `onedir` mode. (#7973)
- (Windows) Fix unintentional randomization of library search path order in the binary dependency analysis step. The incorrect order of search paths would result in defunct builds when using both `pywin32` and `PyQt5`, `PyQt6`, `PySide2`, or `PySide6`, as it would prevent the python's copy of `VCRUNTIME140_1.dll` from being collected into the top-level application directory due to it being shadowed by the Qt-provided copy. Consequently, the application would fail with `ImportError: DLL load failed while importing pywintypes: The specified module could not be found.` (#7978)

- Ensure that `__main__` is always in the list of modules to exclude, to prevent a program or a library that attempts to import `__main__` from pulling PyInstaller itself into frozen application bundle. (#7956)
- Fix *PyInstaller.utils.hooks.collect_entry_point()* so that it returns module names (without class names). This matches the behavior of previous PyInstaller versions that regressed in PyInstaller `v6.0.0` during transition from `pkg_resources` to `importlib.metadata`. (#7958)
- Fix `TypeError: process_collected_binary() got an unexpected keyword argument 'strip'` error when UPX compression is enabled. (#7998)
- Validate binaries returned by analysis of `ctypes` calls in collected modules; the analysis might return files that are in `PATH` but are not binaries, which then cause errors during binary dependency analysis. An example of such problematic case is the `gmsh` package on Windows, where `ctypes.util.find_library('gmsh')` ends up resolving the python script called `gmsh` in the environment's Scripts directory. (#7984)

Hooks

- Update `PySide6.QtHttpServer` hook for compatibility with `PySide6` 6.5.3 on Windows. (#7994)

PyInstaller Core

- (macOS) Lower the severity of a missing `Info.plist` file in a collected macOS .framework bundle from an error to a warning (unless strict collection mode is enabled). While missing `Info.plist` in a collected .framework bundle will cause `codesign` to refuse to sign the generated .app bundle, the user might be interested in building just the POSIX application or may not plan to sign their .app bundle. Fixes building with old `PyQt5` PyPI wheels (< 5.14.1). (#7959)

2.16.7 6.0.0 (2023-09-22)

Features

- (macOS) PyInstaller now attempts to preserve the `.framework` bundles when collecting shared libraries from them. If a shared library is to be collected from a `.framework` bundle, the `Info.plist` is also automatically collected. The `.framework` bundle collection code also attempts to fix the bundles' structure to conform to code-signing requirements (i.e., creation of the `Current` symbolic link in the `Versions` directory, and top-level contents being symbolic links that point to counterparts in the `Versions/Current` directory). Note that other resources (for example from `Resources` or `Helpers` directories) still need to be explicitly collected by hooks. (#7619)
- (macOS) The file relocation mechanism in `BUNDLE` that generates macOS .app bundles has been completely redesigned. All data files are now placed into directory structure rooted in `Contents/Resources`, all shared libraries (as well as nested .framework bundles) are placed into directory structure rooted in `Contents/Frameworks`, and only the the program executable is placed into the `Contents/MacOS` directory. The contents of `Contents/Resources` and `Contents/Frameworks` directories are cross-linked via symlinks between the two directory trees in order to maintain illusion of mixed-content directories (in both directory trees). The cross-linking is done at either file level or (sub)directory level, depending on the content type of a (sub)directory. For directories in `Contents/Frameworks` that contain a dot in the name, a work-around is automatically applied: the directory is created with a modified name that does not include the dot, and next to it, a symbolic link is created under the original name and pointing to the directory with modified name. (#7619)
- (non-Windows) PyInstaller now attempts to preserve the parent directory structure for shared libraries discovered and collected by the binary dependency analysis, instead of automatically collecting them into the top-level

application directory. Due to library search path assumptions made in various places, symbolic links to collected libraries are created in the top-level application directory. This complements earlier work (#7028) that implemented DLL parent directory structure preservation on Windows. (#7619)

- (Windows) Add an option to hide or minimize the console window in console-enabled applications, but only if the program's process owns the console window (i.e., the program was not launched from an existing console window). (#7729)
- (Windows) The *--add-data* and *--add-binary* options accept the POSIX syntax of `--add-data=source:dest` rather than `--add-data=source;dest`. The latter will continue to work on Windows to avoid breaking backwards compatibility but is discouraged in favour of the now cross platform format. (#6724)
- Add automatic binary vs. data file (re)classification step to the analysis process. PyInstaller now inspects all files passed to `Analysis` via `datas` and `binaries` arguments, as well as all files returned by hooks via `datas` and `binaries` hook global variables. The inspection mechanism is platform-specific, and currently implemented for Windows, Linux, and macOS. Proper file classification ensures that all collected binary files undergo binary dependency analysis and any other platform-specific binary processing. On macOS, it also helps ensuring that the collected files are placed in the proper directory in the generated .app bundles. (#7619)
- Add support for specifying hash randomization seed via `hash_seed=<value>` *run-time option* when building the application. This allows the application to use a fixed seed value or disable hash randomization altogether by using seed value of 0. (#7847)
- Allow spec files to take custom command line parameters. See *adding parameters to spec files*. (#4482)
- Extend the operation retry mechanism that was initially introduced by #7840 to cover all processing steps that are performed during assembly of a Windows executable. This attempts to mitigate the interference from anti-virus programs and other security tools, which may temporarily block write access to the executable for a scan between individual processing steps. (#7871)
- Implement pass-through for Python's X-options via PyInstaller's *run-time options mechanism*. (#7847)
- Implement support for creating symbolic links from special `'SYMLINK'` TOC entries, either at build-time (`onedir` mode) or at run-time (`onefile` mode). Implement symbolic link preservation support in the analysis process; if a file and a symbolic link pointing to it are both to be collected, and if their relative relationship is preserved in the frozen application, the symbolic link is collected as a `'SYMLINK'` entry. (#7619)
- Implement *PyInstaller.utils.hooks.check_requirement()* hook utility function as an `importlib.metadata`-based replacement for *PyInstaller.utils.hooks.is_module_satisfies()*; the latter is now just an alias for the former, kept for compatibility with existing hooks. (#7943)
- Restructure onedir mode builds so that everything except the executable (and `.pkg` if you're using external PYZ archive mode) are hidden inside a sub-directory. This sub-directory's name defaults to `_internal` but may be configured with a new *--contents-directory* option. Onefile applications and macOS `.app` bundles are unaffected. (#7713)
- The *PyInstaller.utils.hooks.collect_all()* hook utility function now attempts to translate the given importable package name into distribution name before attempting to collect metadata. This allows the function to handle cases when the distribution name does not match the importable package name. (#7943)

Bugfix

- (macOS) `QtWebEngine` now works in `onefile` builds (previously available only in `onedir` builds). (#4361)
- (macOS) Fix the shared library duplication problem where a shared library that is also referred to via its symbolic links (e.g., a shared library `libwx_baseu-3.1.5.0.0.dylib` with symbolic links `libwx_baseu-3.1.5.dylib` and `libwx_baseu-3.0.dylib`) ends up collected as duplicates and consequently crashes the program. The symbolic links should now be preserved, thus avoiding the problem. (#5710)
- (macOS) In generated .app bundles, the data files from `PySide2`, `PySide6`, `PyQt5`, or `PyQt6` directory are now relocated to the directory structure rooted in `Contents/Resources` to ensure compliance with code-signing requirements. The content cross-linking between `Contents/Resources` and `Contents/Frameworks` should ensure that `QML` components in the `qml` sub-directory continue to work in spite of plugins (shared libraries) being technically separated from their corresponding metadata files. The automatic work-around for directories with dots in names should prevent code-signing issues due to some `QML` components in Qt5 having dot in their names (e.g. `QtQuick.2` and `QtQuick/Controls.2`. (#7619)
- (macOS) In generated .app bundles, the source .py files are now again relocated to `Contents/Resources` directory (and cross-linked into `Contents/Frameworks`), which ensures that code-signing does not store signatures into the files' extended attributes. This reverts the exemption made in #7180 to accommodate the `cv2` loader script; the problem is now solved by cross-linking binaries from `Contents/Frameworks` to `Contents/Resources`, which allows the loader to find the extension binary (or rather, a symbolic link to it) next to the .py file. (#7619)
- (macOS) Sandboxing for `QtWebEngine` in `PySide6` and `PyQt6` is not disabled anymore by the corresponding run-time hooks (see #6903), as it should work out-of-the-box thanks to PyInstaller now preserving the structure of the `QtWebEngineCore.framework` bundle. (#7619)
- (macOS) The main process in a program that uses `QtWebEngine` is not mis-identified as `QtWebEngineCore` anymore in the application's menu bar. This applies to `onedir` POSIX program builds (i.e. the .app bundles were not affected by this). (#5409)
- (Windows) Avoid aborting the build process if machine type (architecture) cannot be determined for a DLL in a candidate search path; instead, skip over such files, and search in other candidate paths. Fixes build errors when a search path contains an invalid DLL file (for example, a stub file). (#7874)
- (Windows) Prevent PyInstaller's binary dependency analysis from looking for shared libraries in all `sys.path` locations. Instead, search only `sys.base_prefix` and `pywin32` directories, of available. This, for example, prevents PyInstaller from picking up incompatible DLLs from system-installed programs that happen to put their installation directory into system-wide `PYTHONPATH`. (#5560)
- (Windows) Remove the use of deprecated `distutils.sysconfig` module. The import of this module seems to cause the python process to crash when `tensorflow` is subsequently imported during import analysis. (#7347)
- Fix file duplication when collecting a file and symbolic links pointing at it; with new symbolic link support, the symbolic links are now properly preserved. This should help reducing the size of builds made on Linux and macOS with Anaconda, which provides versioned symbolic links for packaged shared libraries, and PyInstaller tends to collect them all due to hook helper based on the packages' metadata. (#7619)
- Fix incompatibility between PyInstaller's frozen importer (`PyiFrozenImporter`) and `importlib.resources` when trying to look up the resources of a collected namespace package via `importlib.resources.files()`. (#7921)
- When copying files into `onedir` application bundles, use `shutil.copyfile()` instead of `shutil.copy2()` to avoid issues with original permissions/metadata being too restrictive. (#7938)

Incompatible Changes

- (Linux) Removed support for building LSB-compliant bootloader, due to lack of support for LSB (Linux Standard Base) in contemporary linux distributions. (#7807)
- (macOS) Due to relocation of all dynamic libraries into directory structure rooted in the `Contents/Frameworks` directory, the `sys._MEIPASS` variable as well as the `os.path.dirname(__file__)` in the entry-point script now point to `Contents/Frameworks` instead of `Contents/MacOS`, while `os.path.dirname(sys.executable)` continues to point to the `Contents/MacOS` directory. The behavior change applies only to `onedir` .app bundles (in `onefile` ones, `sys._MEIPASS` and `__file__` of the entry-point script have always pointed to the temporary extraction directory and continue to do so). (#7619)
- (macOS) The changes made to the macOS .app bundle generation code and the resulting .app bundle structure (strict relocation of binaries to `Contents/Frameworks` and data files to `Contents/Resources`, bi-directional cross-linking between `Contents/Frameworks` and `Contents/Resources`, preservation of nested .framework bundles, automatic work-around for dots in directory names) are likely incompatible with existing (external) post-processing scripts. (#7619)
- (Windows) Removed command-line options related to processing of the WinSxS assemblies: `--win-private-assemblies` and `--win-no-prefer-redirects`. The corresponding arguments to `Analysis` are deprecated and raise and error if set to `True`. (#7784)
- (Windows) Removed support for analyzing and collection of dependencies referenced via WinSxS (side-by-side) assemblies. This affects binaries compiled with Visual Studio 2008 and earlier, as VC9 run-time was the last version to make use of WinSxS. If you require support for such binaries and you need referenced WinSxS binaries collected with your application, use older version of PyInstaller. (#7784)
- (Windows) Removed support for external application manifest in onedir builds. Removed the `--no-embed-manifest` command-line option and deprecated the corresponding `embed_manifest` argument to `EXE` to raise an error if set to `False`. (#7784)
- All of onedir build's contents except for the executable are now moved into a sub-directory (called `_internal` by default). `sys._MEIPASS` is adjusted to point to this `_internal` directory. The breaking implications for this are:
  - Assumptions that `os.path.dirname(sys.executable) == sys._MEIPASS` will break. Code locating application resources using `os.path.dirname(sys.executable)` should be adjusted to use `__file__` or `sys._MEIPASS` and any code locating the original executable using `sys._MEIPASS` should use `sys.executable` directly.
  - Any custom post processing steps (either in the `.spec` file or externally) which modify the bundle will likely need adjusting to accommodate the new directory. (#7713)
- PyInstaller-frozen applications are not affected by the `PYTHONUTF8` environment variable anymore. To permanently enable or disable the UTF8 mode, use the `X utf8_mode=1` or `X utf_mode=0` *run-time option* when building the application. (#7847)
- Remove bytecode encryption (`--key` and `cipher` options). (#6999)
- Remove the `--ascii` command-line option, which is an effective no-op under python 3; the `codecs` module is always collected due to being listed among the base modules. (#7801)
- Remove the built-in attempt at collection of data files from packages that are installed as python eggs. Collection of all non-python resources from packages should be handled in the standardized way via hooks, regardless of how a package is installed. (#7784)
- Remove support for zipped eggs. PyInstaller will not collect python code nor resources from zipped eggs, nor will it collect zipped eggs as a whole. (#7784)

- Remove the `requirements_for_package` hook utility function, which was primarily used by *`collect_all()`*; the latter does not include the top-level modules of metadata-declared requirements among the returned hidden imports anymore. (#7943)
- The *`PyInstaller.utils.hooks.collect_data_files()`* hook utility helper does not collect `.pyc` files from `__pycache__` directories anymore, even with `include_py_files=True` argument. (#7943)
- The *`PyInstaller.utils.hooks.is_module_satisfies()`* helper does not support the `version` and `version_attribute` arguments anymore; the function will raise an error if they are specified. If the distribution specified in the `requirements` string is not found, the function will not attempt to import the eponymous module and read its version attribute anymore. (#7943)
- The collection of "py files", enabled by the `include_py_files=True` argument to the *`PyInstaller.utils.hooks.collect_data_files()`* hook utility function, is now restricted to only `.py` and `.pyc` files. Previously, all suffices from `importlib.machinery.all_suffixes()` were enabled, which resulted in spurious collection of dynamic libraries and extensions (due to `.so`, `.abi3.so`, `.pyd`, etc. being among those suffices). (#7943)

Bootloader

- (Linux, macOS) When extracting files from `onefile` archive, the executable bit is now set only on binaries (files whose TOC type code was either `BINARY`, `EXECUTABLE`, or `EXTENSION`) or data files that originally had the executable bit set. Therefore, binaries are now extracted with permissions bits set to `0700`, while all other files have permissions bits set to `0600`. (#7950)
- Use PEP 587 Python Initialization Configuration API to configure the embedded Python interpreter. (#7847)

PyInstaller Core

- (Windows) The temporary/intermediate executable files are not generated with `.notanexecutable` suffix anymore, as the retry mechanism from #7840 and #7871 is now the preferred way of dealing with anti-virus program interference during the build. (#7871)
- Avoid collecting `pathlib` and `tokenize` (and their dependencies, such as `urllib`) into `base_library.zip`. By collecting them into PYZ archive, only submodules that the application really requires can be collected, which helps reducing the size of applications that, for example, do not require the full `urllib` package. (#7836)
- Drop support for end of life Python 3.7. (#7733)

Bootloader build

- To enable the passing of extra arguments to the bootloader compiler during installation via pip, you can utilize the environment variable `PYINSTALLER_BOOTLOADER_WAF_ARGS`. However, it is essential to ensure that the environment variable `PYINSTALLER_COMPILE_BOOTLOADER` is present for this functionality to work correctly. (#7796)

2.16.8 5.13.2 (2023-08-29)

Bugfix

- (Windows) Fix `OSError: exception: access violation reading 0x00000010` raised by `matplotlib` and `win32com` run-time hooks when ran in 32-bit frozen application (regression introduced in `v5.13.1`). (#7893)

Hooks

- Fix the license of the new `_pyi_rth_utils` run-time package; it is now licensed under permissive Apache license, which matches the license of the run-time hooks that use this run-time package. (#7894)

PyInstaller Core

- Fix the license of the `pyi_splash` run-time module; it is now licensed under permissive Apache license to avoid unintentionally imposing additional license restrictions on the frozen applications that make use of this module. (#7896)

2.16.9 5.13.1 (2023-08-26)

Security

- (Windows) Ensure that the access to temporary directories created by the `matplotlib` and `win32com` run-time hooks is restricted to the user running the frozen application, even if the directory in the `TMP` or `TEMP` variables points to a system-wide *world writable* location that can be accessed by all users. (#7827)

Bugfix

- (macOS) Fix `pkgutil.iter_modules()` failing to find submodules of a package that contains data files when running as a macOS .app bundle. (#7884)
- (Windows) Fix `win32com` run-time hook to fully isolate the `gen_py` cache. This prevents access to the global cache, which results in errors when the global cache contains some, but not all, required modules. (#6257)
- (Windows) Fix splash screen not being able to locate collected Tk resources in onefile applications created in MSYS2 python environment. (#7828)
- (Windows) Fixed bug where GdkPixbuf loaders.cache dll paths are absolute paths (e.g. `C:/tools/msys64/mingw64/lib/gdk-pixbuf-2.0/2.10.0/loaders/*.dll`) and not relative paths (e.g. `lib\\gdk-pixbuf\\loaders\\libpixbufloader-png.dll`) when the file is generated in the MSYS2/mingw64 environment. This results in the program crashing when run on another Windows machine because it cannot find the GdkPixbuf loader DLLs. (#7842)
- Exclude NVIDIA graphics driver libraries from vendoring. (#7746)
- Fix error handling in Glib schema compilation helper function. Ignore character encoding errors when reading stdout/stderr from `glib-schema-compile` process; this fixes errors in MSYS2/mingw64 environment, caused by `U+201C` and `U+201D` quotation marks in the output. (#7833)
- Implement a work-around for un-initialized `sys._stdlib_dir` and ensure that python-frozen stdlib modules in Python >= 3.11 have `__file__` attribute set. (#7847)

Hooks

- Add support for commercial PyQt5 and PyQt6 wheels. (#7770)

Bootloader

- Have bootloader call `Py_GetPath()` before `Py_SetPath()` on all platforms (instead of just on Windows) to work around memory-initialization issues in python 3.8 and 3.9, which come to light with `PYTHONMALLOC=debug` or `PYTHONDEVMODE=1` being set in the environment. (#7790)

2.16.10 5.13.0 (2023-06-24)

Features

- Add support for Python 3.12. (#7670)
- (Windows) Add support for collecting .pyc files from the `python3X.zip` archive where `Windows embeddable package` Python stores its stdlib modules. (#4989)

Bugfix

- Limit the import of collected packages prior to performing binary dependency analysis to only Windows, where it is actually useful. On non-Windows platforms, there is no benefit to it, and it might cause issues with particular orders of package imports. (#7698)
- When building PKG for `onedir` build, ensure that `DATA` entries are put into dependencies list instead of including them in the PKG. This complements existing behavior for `BINARY` and `EXTENSION` entries, and prevents a `onedir` build from becoming a broken `onefile` one if user accidentally passes binaries and data files TOCs to `EXE` instead of *COLLECT* when manually editing the spec file. (#7708)

2.16.11 5.12.0 (2023-06-08)

Features

- (macOS) PyInstaller now removes all rpaths from collected binaries and replaces them with a single rpath pointing to the top-level application directory, relative to `@loader_path`. (#7664)
- Attempt to preserve the parent directory layout for `pywin32` extensions that originate from `win32` and `pythonwin` directories, instead of collecting those extensions to top-level application directory. (#7627)

Bugfix

- (Linux/macOS) Fix the Qt directory path override in `PySide2` and `PySide6` run-time hooks. These paths, set via `QT_PLUGIN_PATH` and `QML2_IMPORT_PATH` environment variables, are used with `PySide2` and `PySide6` builds that that use system-wide Qt installation and are not portable by default (e.g. Homebrew). (#7649)
- (macOS) When rewriting the dylib identifier and paths to linked libraries in a collected binary, instead of directly using `@loader_path`-based path, use `@rpath`-based path and replace rpaths in the binary with a single rpath that points to the top-level application directory, relative to `@loader_path`. This ensures that the library identifiers of collected shared libraries and their references in referring binaries always match, which allows packages to pre-load a library from an arbitrary location via for example `ctypes`. (#7664)

- (Windows) Fix string serialization of `VSVersionInfo` to account for the possibility of `StringStruct` values containing quote characters. (#7630)
- Attempt to fix compatibility of PyInstaller's `PyiFrozenImporter` with `importlib.util.LazyLoader`. (#7657)
- Attempt to mitigate issues with Anaconda `pywin32` package that result from the package installing three copies of `pywintypes3X.dll` and `pythoncom3X.dll` in different locations. (#7627)
- Changes made to `datas` and `binaries` lists that are passed to `Analysis` constructor will now invalidate the cached `Analysis` and trigger a re-build. This applies both to changes made by editing the .spec file manually and to automatic changes due to addition or removal of corresponding command-line options (*--add-data*, *--add-binary*, *--collect-data*, *--collect-binaries*, *--copy-metadata*). Previously, changes might not have taken effect as the old cached build was returned if available and unless user explicitly requested a clean build using the *--clean* command-line option. (#7653)
- Ensure that `qt_{lang}` translation files are collected with the `QtCore` module, in addition to already-collected `qtbase_{lang}` files. Applies to all four Qt-bindings: `PySide2`, `PySide6`, `PyQt5`, and `PyQt6`. (#7682)
- Fix `ModuleNotFoundError: No module named 'ipaddress'` for any application built with Python >=3.11.4. (#7692)
- Fix splash-enabled program crashing due to NULL-pointer dereference in the bootloader when the Tcl/Tk shared libraries cannot be loaded. The program should now run the user's python code, where it will raise an exception if the `pyi_splash` module is used. (#7679)
- Implement proper binary dependency scanning in the `SPLASH` target, so that binary dependencies of the Tcl and Tk shared libraries are always collected and added to the list of splash requirements (for pre-extraction in onefile builds). This fixes the splash screen when building with Windows build of python.org Python 3.12b1, which ships Tcl shared library with new dependency on `zlib1.dll`. (#7679)

PyInstaller Core

- (macOS) Use macOS-provided `install_name_tool` utility to modify headers on collected binaries: change the dylib identifier to `@rpath/<name>.dylib`, rewrite paths to linked non-system shared libraries to `@rpath/<dependency>`, remove any additional rpaths and add an rpath pointing to the application's top-level directory, relative to the `@loader_path`. Previously, the header modification was performed using `macholib` and was limited only to modification of dylib identifier and paths to linked non-system shared libraries. (#7664)

2.16.12 5.11.0 (2023-05-13)

Features

- Add a work-around for pure-python modules that do not specify encoding via **PEP 263** encoding header but contain non-ASCII characters in local (non-UTF8) encoding. When such characters are present only in code comments, python still loads and runs the module, but attempting to retrieve its source code via the loader's `get_source()` method results in a `UnicodeDecodeError`, which interrupts the analysis process. The error is now caught and a fall-back codepath attempts to retrieve the source code as raw data to avoid encoding issues. (#7622)

Bugfix

- (Windows) Avoid writing collected binaries to binary cache unless they need to be processed (i.e., only if binary stripping or `upx` processing is enabled). (#7595)
- Fix a regression in bootloader that caused crash in onefile executables when encountering a duplicated entry in the PKG/CArchive and the `PYINSTALLER_STRICT_UNPACK_MODE` environment variable not being set. (#7613)

Deprecations

- The `TOC` class is now deprecated; use a plain `list` with the same three-element tuples instead. PyInstaller now performs explicit normalization (i.e., entry de-duplication) of the TOC lists passed to the build targets (e.g., `PYZ`, `EXE`, `COLLECT`) during their instantiation. (#7615)

Bootloader

- Fix bootloader building with old versions of `gcc` that do not support the `-Wno-error=unused-but-set-variable` compiler flag (e.g., `gcc` v4.4.3). (#7592)

Documentation

- Update the documentation on TOC lists and `Tree` class to reflect the deprecation of the `TOC` class. (#7615)

PyInstaller Core

- Remove the use of the `TOC` class in the analysis / build process, and use plain `list` instances instead. The implicit normalization (de-duplication) of TOC entries performed by the `TOC` class has been replaced with explicit normalization. The TOC lists produced by `Analysis` are explicitly normalized at the end of Analysis instantiation, before they are stored in the Analysis properties (e.g., `Analysis.pure`, `Analysis.binaries`, `Analysis.datas`). Similarly, the TOC lists passed to the build targets (e.g., `PYZ`, `EXE`, `COLLECT`) are explicitly normalized as part of the targets' instantiation process. (#7615)

2.16.13 5.10.1 (2023-04-14)

Bugfix

- Fix regression on platforms with strict data alignment requirements (such as linux on `armhf`/`armv7`), caused by bug in PKG/CArchive generation that was introduced during the archive writer code cleanup. The regression caused executable to terminate with `Bus error` on the affected platforms, such as 32-bit Debian Buster on Raspberry Pi 4. (#7566)

2.16.14 5.10.0 (2023-04-11)

Bugfix

- (Linux) Ignore the executable name resolution based on `/proc/self/exe` when the PyInstaller-frozen executable is launched via the `ld.so` dynamic loader executable. In such cases, the resolved name points to the `ld.so` executable, causing the PyInstaller-frozen executable to fail with *Cannot open PyInstaller archive from executable…* error. (#7551)
- Ensure that binaries that are manually specified in the .spec file (or via corresponding *`--add-binary`* or *`--collect-binaries`* command-line switches) undergo the binary dependency analysis, so their dependencies are automatically collected. (#7522)
- Extend the `excludedimports` mechanism rework from #7066 to properly handle relative imports within the package. For example, ensure that `excludedimports = ['a.b']` within the hook for package `a` takes effect when package `a` does `from . import b` (in addition to `from a import b`). (#7495)
- Extend the `excludedimports` mechanism rework from #7066 to properly handle the case of multiple submodules being imported in a single `from ... import ...` statement (using absolute or relative import). For example, when package `c` does `from d import e, f`, we need to consider potential `excludedimports` rules matching package `d` and, if `d` itself is not excluded, potential rules individually matching `d.e` and `d.f`. (#7495)
- Fix marshal error in binary dependency search stage, caused by the list of collected packages containing a `modulegraph.Alias` instance instead of only plain `str` instances. (#7515)
- Reorganize the `multiprocessing` run-time hook to override `Popen` implementations only for `spawn` and `forkserver` start methods, but not for the `fork` start method. This avoids a dead-lock when attempting to perform nested multiprocessing using the `fork` start method, which occurred due to override-provided lock (introduced in #7411) being copied in its locked state into the forked sub-process. (#7494)

Incompatible Changes

- The `archive_viewer` utility has been rewritten with modified command-line interface (`--log` has been renamed to `--list`) and with changed output formatting. (#7518)

Hooks

- (Windows) Improve support for `matplotlib >= 3.7.0` by collecting all `delvewheel`-generated files from the `matplotlib.libs` directory, including the load-order file. This is required when PyPI `matplotlib` wheels are used in combination with Anaconda python 3.8 and 3.9. (#7503)
- Add hook for `PyQt6.QtSpatialAudio` module, which was added in `PyQt6` 6.5.0. (#7549)
- Add hook for `PyQt6.QtTextToSpeech` module, which was added in `PyQt6` 6.4 series. (#7549)
- Extend `PySide6` hooks for `PySide6` 6.5.0 compatibility: add hooks for `QtLocation`, `QtTextToSpeech`, and `QtSerialBus` modules that were introduced in `PySide` 6.5.0. (#7549)

Documentation

- Clarify the supported color specification formats and apply consistent formatting of default parameter values in the splash screen documentation. (#7529)

2.16.15 5.9.0 (2023-03-13)

Features

- Choose *hooks provided by packages* over hooks from pyinstaller-hooks-contrib if both provide the same hook. (#7456)

Bugfix

- Fix changes to `sys.path` made in the spec file being ignored by hook utility functions (e.g. *`collect_submodules()`*). (#7456)

2.16.16 5.8.0 (2023-02-11)

Features

- Compile the collected GLib schema files using `glib-schema-compiler` instead of collecting the pre-compiled `gschemas.compiled` file, in order to properly support collection of schema files from multiple locations. Do not collect the source schema files anymore, as only `gschemas.compiled` file should be required at run time. (#7394)

Bugfix

- (Cygwin) Avoid using Windows-specific codepaths that require `pywin32-ctypes` functionality that is not available in Cygwin environment. (#7382)
- (non-Windows) Fix race condition in environment modification done by `multiprocessing` runtime hook when multiple threads concurrently spawn processes using the `spawn` method. (#7410)
- (Windows) Changes in the version info file now trigger rebuild of the executable file. (#7338)
- Disallow empty source path in the `binaries` and `datas` tuples that are returned from the hooks and sanitized in the `PyInstaller.building.utils.format_binaries_and_datas`. The empty source path is usually result of an error in the hook's path retrieval code, and causes implicit collection of the whole current working directory. This is never the intended behavior, so raise a `SystemExit`. (#7384)
- Fix *unknown log level* error raised with `--log-level=DEPRECATION`. (#7413)

Incompatible Changes

- The deprecated PEP-302 `find_module()` and `load_module()` methods have been removed from PyInstaller's `FrozenImporter`. These methods have not been used by python's import machinery since python 3.4 and PEP-451, and were effectively left untested and unmaintained. The removal affects 3rd party code that still relies on PEP-302 finder/loader methods instead of the PEP-451 ones. (#7344)

Hooks

- Collect `multimedia` plugins that are required by `QtMultimedia` module starting with Qt6 v6.4.0. (#7352)
- Do not collect `designer` plugins as part of `QtUiTools` module in `PySide2` and `PySide6` bindings. Instead, tie the collection of plugins only to the `QtDesigner` module. (#7322)

Module Loader

- Remove deprecated PEP-302 functionality from `FrozenImporter`. The `find_module()` and `load_module()` methods are deprecated since python 3.4 in favor of PEP-451 loader. (#7344)

2.16.17 5.7.0 (2022-12-04)

Features

- Add the package's location and exact interpreter path to the error message for the check for obsolete and PyInstaller-incompatible standard library back-port packages (`enum34` and `typing`). (#7221)
- Allow controlling the build log level (*`--log-level`*) via a `PYI_LOG_LEVEL` environment variable. (#7235)
- Support building native ARM applications for Windows. If PyInstaller is ran on an ARM machine with an ARM build of Python, it will prodice an ARM application. (#7257)

Bugfix

- (Anaconda) Fix the `PyInstaller.utils.hooks.conda.collect_dynamic_libs` hook utility function to collect only dynamic libraries, by introducing an additional type check (to exclude directories and symbolic links to directories) and additional suffix check (to include only files whose name matches the following patterns: `*.dll`, `*.dylib`, `*.so`, and `*.so.*`). (#7248)
- (Anaconda) Fix the problem with Anaconda python 3.10 on linux and macOS, where all content of the environment's `lib` directory would end up collected as data due to additional symbolic link pointing from `python3.1` to `python3.10`. (#7248)
- (GNU/Linux) Fixes an issue with gi shared libraries not being packaged if they don't have version suffix and are in a special location set by `LD_LIBRARY_PATH` instead of a typical library path. (#7278)
- (Windows) Fix the problem with `windowed` frozen application being unable to spawn interactive command prompt console via `subprocess` module due to interference of the `subprocess` runtime hook with stream handles. (#7118)
- (Windows) In `windowed/noconsole` mode, stop setting `sys.stdout` and `sys.stderr` to custom `NullWriter` object, and instead leave them at `None`. This matches the behavior of windowed python interpreter (`pythonw.exe`) and prevents interoperability issues with code that (rightfully) expects the streams to be either `None` or objects that are fully compatible with `io.IOBase`. (#3503)

- Ensure that `PySide6.support.deprecated` module is collected for `PySide6` 6.4.0 and later in order to enable continued support for `|` and `&` operators between Qt key and key modifier enum values (e.g., `QtCore.Qt.Key_D` and `QtCore.Qt.AltModifier`). (#7249)
- Fix potential duplication of python extension modules in `onefile` builds, which happened when an extension was collected both as an `EXTENSION` and as a `DATA` (or a `BINARY`) TOC type. This resulted in run-time warnings about files already existing; the most notorious example being `WARNING: file already exists but should not: C:\Users\user\AppData\Local\Temp\MEI1234567\torch\_C.cp39-win_amd64.pyd` when building `onefile` applications that use `torch`. (#7273)
- Fix spurious attempt at reading the `top_level.txt` metadata from packages installed in egg form. (#7086)
- Fix the log level (provided via *`--log-level`*) being ignored by some build steps. (#7235)
- Fix the problem with `MERGE` not properly cleaning up passed `Analysis.binaries` and `Analysis.datas` TOCs due to changes made to `TOC` class in PyInstaller 5.0. This effectively broke the supposed de-duplication functionality of `MERGE` and multi-package bundles, which should be restored now. (#7273)
- Prevent `$pythonprefix/bin` from being added to `sys.path` when PyInstaller is invoked using `pyinstaller your-code.py` but not using `python -m PyInstaller your-code.py`. This prevents collection mismatch when a library has the same name as console script. (#7120)
- Prevent isolated-subprocess calls from indefinitely blocking in their clean-up codepath when the subprocess fails to exit. After the grace period of 5 seconds, we now attempt to terminate such subprocess in order to prevent hanging of the build process. (#7290)

Incompatible Changes

- (Windows) In `windowed/noconsole` mode, PyInstaller does not set `sys.stdout` and `sys.stderr` to custom `NullWriter` object anymore, but leaves them at `None`. The new behavior matches that of the windowed python interpreter (`pythonw.exe`), but may break the code that uses `sys.stdout` or `sys.stderr` without first checking that they are available. The code intended to be run frozen in `windowed/noconsole` mode should be therefore be validated using the windowed python interpreter to catch errors related to console being unavailable. (#7216)

Deprecations

- Deprecate bytecode encryption (the `--key` option), to be removed in PyInstaller v6.0. (#6999)

Hooks

- (Windows) Remove the `subprocess` runtime hook. The problem with invalid standard stream handles, which caused the `subprocess` module raise an `OSError: [WinError 6] The handle is invalid` error in a `windowed onefile` frozen application when trying to spawn a subprocess without redirecting all standard streams, has been fixed in the bootloader. (#7182)
- Ensure that each `Qt*` submodule of the `PySide2`, `PyQt5`, `PySide6`, and `PyQt6` bindings has a corresponding hook, and can therefore been imported in a frozen application on its own. Applicable to the latest versions of packages at the time of writing: `PySide2 == 5.15.2.1`, `PyQt5 == 5.15.7`, `PySide6 == 6.4.0`, and `PyQt6 == 6.4.0`. (#7284)
- Improve compatibility with contemporary `Django` 4.x version by removing the override of `django.core.management.get_commands` from the `Django` run-time hook. The static command list override is both outdated (based on `Django` 1.8) and unnecessary due to dynamic command list being properly populated under contemporary versions of `PyInstaller` and `Django`. (#7259)

- Introduce additional log messages to `matplotlib.backend` hook to provide better insight into what backends are selected and why when the detection of `matplotlib.use` calls comes into effect. (#7300)

Bootloader

- (Windows) In a `onefile` application, avoid passing invalid stream handles (the `INVALID_HANDLE_VALUE` constant with value `-1`) to the launched application child process when the standard streams are unavailable (for example, in a windowed/no-console application). (#7182)

Bootloader build

- Support building ARM native binaries using MSVC using the command `python waf --target-arch=64bit-arm all`. If built on an ARM machine, `--target-arch=64bit-arm` is the default. (#7257)
- Windows ARM64 bootloaders may now be built using an ARM build of clang with `python waf --target-arch=64bit-arm --clang all`. (#7257)

2.16.18 5.6.2 (2022-10-31)

Bugfix

- (Linux, macOS) Fix the regression in shared library collection, where the shared library would end up collected under its fully-versioned .so name (e.g., `libsomething.so.1.2.3`) instead of its originally referenced name (e.g., `libsomething.so.1`) due to accidental symbolic link resolution. (#7189)

2.16.19 5.6.1 (2022-10-25)

Bugfix

- (macOS) Fix regression in macOS app bundle signing caused by a typo made in #7180. (#7184)

2.16.20 5.6 (2022-10-23)

Features

- Add official support for Python 3.11. (Note that PyInstaller v5.5 is also expected to work but has only been tested with a pre-release of Python 3.11.) (#6783)
- Implement a new hook utility function, *`collect_delvewheel_libs_directory()`*, intended for dealing with external shared library in `delvewheel`-enabled PyPI wheels for Windows. (#7170)

Bugfix

- (macOS) Fix OpenCV (`cv2`) loader error in generated macOS .app bundles, caused by the relocation of package's source .py files. (#7180)
- (Windows) Improve compatibility with `scipy` 1.9.2, whose Windows wheels switched to `delvewheel`, and therefore have shared libraries located in external .libs directory. (#7168)
- (Windows) Limit the DLL parent path preservation behavior from #7028 to files collected from site-packages directories (as returned by `site.getsitepackages()` and `site.getusersitepackages()`) instead of all paths in `sys.path`, to avoid unintended behavior in corner cases, such as `sys.path` containing the drive root or user's home directory. (#7155)
- Fix compatibility with `PySide6` 6.4.0, where the deprecated `Qml2ImportsPath` location key is not available anymore; use the new `QmlImportsPath` key when it is available. (#7164)
- Prevent PyInstaller runtime hook for `setuptools` from attempting to override `distutils` with `setuptools`-provided version when `setuptools` is collected and its version is lower than 60.0. This both mimics the unfrozen behavior and prevents errors on versions between 50.0 and 60.0, where we do not explicitly collect `setuptools._distutils`. (#7172)

Incompatible Changes

- (macOS) In generated macOS .app bundles, the collected source .py files are not relocated from `Contents/MacOS` to `Contents/Resources` anymore, to avoid issues when the path to a .py file is supposed to resolve to the same directory as adjacent binary extensions. On the other hand, this change might result in regressions w.r.t. bundle signing and/or notarization. (#7180)

Bootloader

- (Windows) Update the bundled `zlib` sources to v1.2.13. (#7166)

2.16.21 5.5 (2022-10-08)

Features

- (Windows) Support embedding multiple icons in the executable. (#7103)

Bugfix

- (Windows) Fix a regression introduced in PyInstaller 5.4 (#6925), where incorrect copy of `python3.dll` (and consequently an additional, incorrect copy of `python3X.dll` from the same directory) is collected when additional python installations are present in `PATH`. (#7102)
- (Windows) Provide run-time override for `ctypes.util.find_library` that searches `sys._MEIPASS` in addition to directories specified in `PATH`. (#7097)
- Fix the problem with `pywin32` DLLs not being found when importing `pywin32` top-level extension modules, caused by the DLL directory structure preservation behavior introduced in #7028. Introduce a new bootstrap/loader module that adds the `pywin32_system32` directory, if available, to both `sys.path` and the DLL search paths, in lieu of having to provide a runtime hook script for every single top-level extension module from `pywin32`. (#7110)

Hooks

- Fix an error raised by the `matplotlib.backends` hook when trying to specify the list of backends to collect via the hooks configuration. (#7091)

2.16.22 5.4.1 (2022-09-11)

Bugfix

- (Windows) Fix run-time error raised by `pyi_rth_win32comgenpy`, the run-time hook for `win32com`. (#7079)

2.16.23 5.4 (2022-09-10)

Features

- (Windows) When collecting a DLL that was discovered via link-time dependency analysis of a collected binary/extension, attempt to preserve its parent directory structure instead of collecting it into application's top-level directory. This aims to preserve the parent directory structure of DLLs bundled with python packages in PyPI wheels, while the DLLs collected from system directories (as well as from `Library\bin` directory of the Anaconda's environment) are still collected into top-level application directory. (#7028)
- Add support for `setuptools`-provided `distutils`, available since `setuptools >= 60.0`. (#7075)
- Implement a generic file filtering decision function for use in hooks, based on the source filename and optional inclusion and exclusion pattern list (*`PyInstaller.utils.hooks.include_or_exclude_file()`*). (#7040)
- Rework the module exclusion mechanism. The excluded module entries, specified via `excludedimports` list in the hooks, are now used to suppress module imports from corresponding nodes *during modulegraph construction*, rather than to remove the nodes from the graph as a post-processing step. This should make the module exclusion more robust, but the main benefit is that we avoid running (potentially many and potentially costly) hooks for modules that would end up excluded anyway. (#7066)

Bugfix

- (Windows) Attempt to extend DLL search paths with directories found in the *PATH* environment variable and by tracking calls to the *os.add_dll_directory* function during import of the packages in the isolated sub-process that performs the binary dependency scanning. (#6924)
- (Windows) Ensure that ANGLE DLLs (`libEGL.dll` and `libGLESv2.dll`) are collected when using Anaconda-installed `PyQt5` and `Qt5`. (#7029)
- Fix `AssertionError` during build when analysing a `.pyc` file containing more that 255 variable names followed by an import statement all in the same namespace. (#7055)

Incompatible Changes

- (Windows) PyInstaller now attempts to preserve parent directory structure of DLLs that are collected from python packages (e.g., bundled with packages in PyPI wheels) instead of collecting them to the top-level application directory. This behavior might be incompatible with 3rd party hooks that assume the old behavior, and may result in duplication of DLL files or missing DLLs in hook-provided runtime search paths. (#7028)

Hooks

- Implement new `gstreamer` hook configuration group with `include_plugins` and `exclude_plugins` options that enable control over GStreamer plugins collected by the `gi.repository.Gst` hook. (#7040)
- Provide hooks for additional `gstreamer` modules provided via GObject introspection (`gi`) bindings: `gi.repository.GstAllocators`, `gi.repository.GstApp`, `gi.repository.GstBadAudio`, `gi.repository.GstCheck`, `gi.repository.GstCodecs`, `gi.repository.GstController`, `gi.repository.GstGL`, `gi.repository.GstGLEGL`, `gi.repository.GstGLWayland`, `gi.repository.GstGLX11`, `gi.repository.GstInsertBin`, `gi.repository.GstMpegts`, `gi.repository.GstNet`, `gi.repository.GstPlay`, `gi.repository.GstPlayer`, `gi.repository.GstRtp`, `gi.repository.GstRtsp`, `gi.repository.GstRtspServer`, `gi.repository.GstSdp`, `gi.repository.GstTranscoder`, `gi.repository.GstVulkan`, `gi.repository.GstVulkanWayland`, `gi.repository.GstVulkanXCB`, and `gi.repository.GstWebRTC`. (#7074)

2.16.24 5.3 (2022-07-30)

Features

- (Windows) Implement handling of console control signals in the `onefile` bootloader parent process. The implemented handler suppresses the `CTRL_C_EVENT` and `CTRL_BREAK_EVENT` to let the child process deal with them as they see it fit. In the case of `CTRL_CLOSE_EVENT`, `CTRL_LOGOFF_EVENT`, or `CTRL_SHUTDOWN_EVENT`, the handler attempts to delay the termination of the parent process in order to buy time for the child process to exit and for the main thread of the parent process to clean up the temporary directory before exiting itself. This should prevent the temporary directory of a `onefile` frozen application being left behind when the user closes the console window. (#6591)
- Implement a mechanism for controlling the collection mode of modules and packages, with granularity ranging from top-level packages to individual sub-modules. Therefore, the hooks can now specify whether the hooked package should be collected as byte-compiled .pyc modules into embedded PYZ archive (the default behavior), or as source .py files collected as external data files (without corresponding modules in the PYZ archive). (#6945)

Bugfix

- (non-Windows) Avoid generating debug messages in POSIX signal handlers, as the functions involved are generally not signal-safe. Should also fix the endless spam of `SIGPIPE` that ocurrs under certain conditions when shutting down the frozen application on linux. (#5270)
- (non-Windows) If the child process of a `onefile` frozen application is terminated by a signal, delay re-raising of the signal in the parent process until after the clean up has been performed. This prevents `onefile` frozen applications from leaving behind their unpacked temporary directories when either the parent or the child process is sent the `SIGTERM` signal. (#2379)
- When building with `noarchive=True` (e.g., `--debug noarchive` or `--debug all`), PyInstaller no longer pollutes user-writable source locations with its `.pyc` or `.pyo` files written next to the corresponding source files. (#6591)

- When building with `noarchive=True` (e.g., `--debug noarchive` or `--debug all`), the source paths are now stripped from the collected .pyc modules, same as if PYZ archive was used. (#6591)

Hooks

- Add PyGObject hook for `gi.repository.freetype2`. Remove warning for hidden import not found for gi._gobject with PyGObject 3.25.1+. (#6951)
- Remove `pkg_resources` hidden imports that aren't available including `py2_warn`, `markers`, and `_vendor.pyparsing.diagram`. (#6952)

Documentation

- Document the signal handling behavior Windows and various quirks related to the frozen application shutdown via the Task Manager. (#6935)

2.16.25 5.2 (2022-07-08)

Features

- Detect if an icon file (`.ico` or `.icns`) is of another image type but has been mislabelled as a native icon type via its file suffix then either normalise to a genuinely native image type if `pillow` is installed or raise an error. (#6870)
- Exit gracefully with an explanatory `SystemExit` if the user moves or deletes the application whilst it's still running. Note that this is only detected on trying to load a module which has not already been loaded. (#6856)
- Implement new standard hook variable, called `warn_on_missing_hiddenimports`. This optional boolean flag allows a hook to opt out from warnings generated by missing hidden imports originating from that hook. (#6914)

Bugfix

- (Linux) Fix potential mismatch between the collected Python shared library name and the name expected by the bootloader when using Anaconda environment. The mismatch would occur on some attempts to freeze a program that uses an extension that is also linked against the python shared library. (#6831)
- (Linux) Fix the missing `gi.repository` error in an application frozen on RHEL/Fedora linux with GObject introspection installed from the distribution's RPM package. (#6780)
- (macOS) The `QtWebEngine` hook now makes `QtOpenGL` and `QtDBus` available to the renderer process with framework installs of Qt 6. (#6892)
- (Windows) Optimize EXE PE headers fix-up process in an attempt to reduce the processing time and the memory footprint with large onefile builds. (#6874)
- Add a try/except guard around `ctypes.util.find_library()` to protect against CPython bug #93094 which leads to a `FileNotFoundError`. (#6864)
- Fix regression in PyInstaller v5 where an import of a non-existent GObject introspection (*gi*) module (for example, an optional dependency) in the program causes a build-time error and aborts the build process. (#6897)
- If passed a name of an importable module instead of a package, the *`PyInstaller.utils.hooks.collect_submodules()`* function now returns a list containing the module's name, same as it would for a package without submodules. (#6850)

- Prevent *PyInstaller.utils.hooks.collect_submodules()* from recursing into sub-packages that are excluded by the function passed via the `filter` argument. (#6846)
- The *PyInstaller.utils.hooks.collect_submodules()* function now excludes un-importable subpackages from the returned modules list. (#6850)

Hooks

- (macOS) Disable `QtWebEngine` sandboxing for Qt6 in the corresponding `PySide6` and `PyQt6` run-time hooks as a work-around for the `QtWebEngineProcess` helper process crashing. This is required as of Qt 6.3.1 due to the way PyInstaller collects Qt libraries, but is applied regardless of the used Qt6 version. If you are using an older version of Qt6 and would like to keep the sandboxing, reset the `QTWEBENGINE_DISABLE_SANDBOX` environment variable at the start of your program, before importing Qt packages. (#6903)
- Add support for GTK4 by adding dependencies and updating `gi.repository.Gtk` and `gi.repository.Gdk` to work with `module-versions` in hooksconfig for `gi`. (#6834)
- Refactor the GObject introspection (`gi`) hooks so that the processing is performed only in hook loading stage or in the `hook()` function, but not in the mixture of two. (#6901)
- Update the GObject introspection (`gi`) hooks to use newly-introduced `GiModuleInfo` object to:
  - Check for module availability.
  - Perform typelib data collection; equivalent of old `get_gi_typelibs` function call.
  - Obtain associated shared library path, equivalent of old `get_gi_libdir` function call.

  The `get_gi_typelibs` and `get_gi_libdir` functions now internally use `GiModuleInfo` to provide backwards-compatibility for external users. (#6901)

2.16.26 5.1 (2022-05-17)

Bugfix

- (Windows) Fix the regression causing the (relative) spec path ending up prepended to relative icon path twice, resulting in icon not being found. (#6788)
- Prevent collection of an entire Python site when using *collect_data_files()* or *collect_dynamic_libs()* for single-file modules (#6789)
- Prevent the hook utility functions, such as *collect_submodules()*, *collect_data_files()*, and *collect_dynamic_libs()*, from failing to identify a package when its PEP451-compliant loader does not implement the optional `is_package` method. (#6790)
- The *get_package_paths()* function now supports PEP420 namespace packages - although for backwards-compatibility reasons, it returns only the first path when multiple paths are present. (#6790)
- The hook utility functions *collect_submodules()*, *collect_data_files()*, and *collect_dynamic_libs()*) now support collection from PEP420 namespace packages. (#6790)
- The user-provided spec file path and paths provided via *--workpath* and *--distpath* are now resolved to absolute full paths before being passed to PyInstaller's internals. (#6788)

Hooks

- Exclude `doctest` in the `pickle` hook. Update `PySide2`, `PySide6`, `PyQt5`, and `PyQt6` hooks with hidden imports that were previously pulled in by `doctest` (that was in turn pulled in by `pickle`). (#6797)

Bootloader

- (Windows) Update the bundled `zlib` sources to v1.2.12. (#6804)

Bootloader build

- Building on Windows with MSVC no longer falls to bits if the PyInstaller repo is stored in a directory with a long path. (#6806)

2.16.27 5.0.1 (2022-04-25)

Bugfix

- (Linux) Have `glib` runtime hook prepend the frozen application's data dir to the `XDG_DATA_DIRS` environment variable instead of completely overwriting it. This should fix the case when `xdg-open` is used to launch a system-installed application (for example, opening an URL in a web browser via the `webbrowser` module) and no registered applications being found. (#3668)
- Prevent unactionable errors raised by UPX from terminating the build. (#6757)
- Restore the pre PyInstaller 5.0 behavior of resolving relative paths to icons as relative to the spec file rather than the current working directory. (#6759)
- (Windows) Update system DLL inclusion list to allow collection of DLLs from Visual Studio 2012 (VC11) runtime and Visual Studio 2013 (VC12) runtime, as well as the latest version of Visual Studio 2015/2017/2019/2022 (VC14) runtime (14.3). (#6778)

Hooks

- Refactor `QtWebEngine` hooks to support both pure Widget-based and pure QML/Quick-based applications. (#6753)
- Update PySide6 and PyQt6 hooks for compatibility with Qt 6.3. `QtWebEngine` on Windows and Linux does not provide the `qt.conf` file for the helper executable anymore, so we generate our own version of the file in order for `QtWebengine` -based frozen applications to work. (#6769)

2.16.28 5.0 (2022-04-15)

Features

- (macOS) App bundles built in `onedir` mode can now opt-in for *argv emulation* so that file paths passed from the UI (*Open with...*) are reflected in `sys.argv`. (#5908)
- (macOS) App bundles built in `onedir` mode can now opt-in for *argv emulation* so that file paths received in initial drag & drop event are reflected in `sys.argv`. (#5436)
- (macOS) The *argv emulation* functionality is now available as an optional feature for app bundles built in either `onefile` or `onedir` mode. (#6089)

- (Windows) Embed the manifest into generated `onedir` executables by default, in order to avoid potential issues when user renames the executable (e.g., the manifest not being found anymore due to activation context caching when user renames the executable and attempts to run it before also renaming the manifest file). The old behavior of generating the external manifest file in `onedir` mode can be re-enabled using the `--no-embed-manifest` command-line switch, or via the `embed_manifest=False` argument to `EXE()` in the .spec file. (#6223)
- (Wine) Prevent collection of Wine built-in DLLs (in either PE-converted or fake/placeholder form) when building a Windows frozen application under Wine. Display a warning for each excluded Wine built-in DLL. (#6149)
- Add a *PyInstaller.isolated* submodule as a safer replacement to *PyInstaller.utils.hooks.exec_statement()*. (#6052)
- Improve matching of UPX exclude patterns to include OS-default case sensitivity, the wildcard operator (*), and support for parent directories in the pattern. Enables use of patterns like `"Qt*.dll"` and `"PySide2*.pyd"`. (#6161)
- Make the error handing of *collect_submodules()* configurable. (#6052)

Bugfix

- (macOS) Fix potential loss of Apple Events during `onefile` app bundle start-up, when the child process is not yet ready to receive events forwarded by the parent process. (#6089)
- (Windows) Remove the attempt to load the manifest of a `onefile` frozen executable via the activation context, which fails with *An attempt to set the process default activation context failed because the process default activation context was already set.* message that can be observed in debug builds. This approach has been invalid ever since #3746 implemented direct manifest embedding into the `onefile` executable. (#6203)
- Fix an import leak when *PyInstaller.utils.hooks.get_module_file_attribute()* is called with a sub-module or a sub-package name. (#6169)
- Fix an import leak when *PyInstaller.utils.hooks.is_package()* is called with a sub-module or a sub-package name. (#6169)
- Fix import errors when calling `get_gi_libdir()` during packaging of GTK apps. Enable CI tests of GTK by adding PyGObject dependencies for the Ubuntu builds. (#6300)
- Issue an error report if a *.spec* file will not be generated, but command-line options specific to that functionality are given. (#6660)
- Prevent `onefile` cleanup from recursing into symlinked directories and just remove the link instead. (#6074)

Incompatible Changes

- (macOS) App bundles built in `onefile` mode do not perform *argv emulation* by default anymore. The functionality of converting initial open document/URL events into `sys.argv` entries must now be explicitly opted-in, via `argv_emulation=True` argument to `EXE()` in the .spec file or via *--argv-emulation* command-line flag. (#6089)
- (Windows) By default, manifest is now embedded into the executable in `onedir` mode. The old behavior of generating the external manifest file can be re-enabled using the `--no-embed-manifest` command-line switch, or via the `embed_manifest=False` argument to `EXE()` in the .spec file. (#6223)
- Issue an error report if a *.spec* file will not be generated, but command-line options specific to that functionality are given. (#6660)
- The *PyInstaller.utils.hooks.get_module_attribute()* function now returns the actual attribute value instead of its string representation. The external users (e.g., 3rd party hooks) of this function must adjust their handling of the return value accordingly. (#6169)

- The `matplotlib.backends` hook no longer collects all available `matplotlib` backends, but rather tries to auto-detect the used backend(s) by default. The old behavior can be re-enabled via the *hook configuration option*. (#6024)

Hooks

- Rework the `matplotlib.backends` hook to attempt performing auto-detection of the used backend(s) instead of collecting all available backends. Implement *hook configuration option* that allows users to switch between this new behavior and the old behavior of collecting all backends, or to manually specify the backend(s) to be collected. (#6024)

Bootloader

- Change the behaviour of the `--no-universal2` flag so that it now assumes the target architecture of the compiler (which may be overridden via the `CC` environment variable to facilitate cross compiling). (#6096)
- Refactor Apple Events handling code and move it into a separate source file. (#6089)

Documentation

- Add a *new section* describing Apple Event forwarding behavior on macOS and the optional *argv emulation* for macOS app bundles, along with its caveats. (#6089)
- Update documentation on using `UPX`. (#6161)

PyInstaller Core

- Drop support for Python 3.6. (#6475)

Bootloader build

- (Windows) Enable Control Flow Guard for the Windows bootloader. (#6136)

2.16.29 4.10 (2022-03-05)

Features

- (Wine) Prevent collection of Wine built-in DLLs (in either PE-converted or fake/placeholder form) when building a Windows frozen application under Wine. Display a warning for each excluded Wine built-in DLL. (#6622)

Bugfix

- (Linux) Remove the timeout on `objcopy` operations to prevent wrongful abortions when processing large executables on slow disks. (#6647)
- (macOS) Limit the strict architecture validation for collected binaries to extension modules only. Fixes architecture validation errors when a `universal2` package has its multi-arch extension modules' arch slices linked against distinct single-arch thin shared libraries, as is the case with `scipy` 1.8.0 macOS `universal2` wheel. (#6587)

- (macOS) Remove the 60 seconds timeout for each `codesign` and `lipo` operation which caused build abortion when processing huge binaries. (#6644)
- (Windows) Use a made up (not `.exe`) suffix for intermediate executable files during the build process to prevent antiviruses from attempting to scan the file whilst PyInstaller is still working on it leading to a `PermissionError` at build time. (#6467)
- Fix an attempt to collect a non-existent `.pyc` file when the corresponding source `.py` file has `st_mtime` set to zero. (#6625)

Hooks

- Add `IPython` to the list of excluded packages in the `PIL` hook in order to prevent automatic collection of `IPython` when it is not imported anywhere else. This in turn prevents whole `matplotlib` being automatically pulled in when using `PIL.Image`. (#6605)

Bootloader

- Fix detection of 32-bit `arm` platform when Thumb instruction set is enabled in the compiler. In this case, the `ctx.env.DEST_CPU` in `waf` build script is set to `thumb` instead of `arm`. (#6532)

2.16.30 4.9 (2022-02-03)

Bugfix

- Add support for external paths when running `pkgutil.iter_modules`. Add support for multiple search paths to `pkgutil.iter_modules`. Correctly handle `pkgutil.iter_modules` with an empty list. (#6529)
- Fix finding `libpython3x.so` when Python is installed with pyenv and the python executable is not linked against `libpython3x.so`. (#6542)
- Fix handling of symbolic links in the path matching part of the PyInstaller's `pkgutil.iter_modules` replacement/override. (#6537)

Hooks

- Add hooks for `PySide6.QtMultimedia` and `PyQt6.QtMultimedia`. (#6489)
- Add hooks for `QtMultimediaWidgets` of all four supported Qt bindings (`PySide2`, `PySide6`, `PyQt5`, and `PySide6`). (#6489)
- Add support for `setuptools 60.7.1` and its vendoring of `jaraco.text` in `pkg_resources`. Exit with an error message if `setuptools 60.7.0` is encountered due to incompatibility with PyInstaller's loader logic. (#6564)
- Collect the `QtWaylandClient`-related plugins to enable Wayland support in the frozen applications using any of the four supported Qt bindings (`PySide2`, `PyQt5`, `PySide6`, and `PyQt6`). (#6483)
- Fix the issue with missing `QtMultimediaWidgets` module when using `PySide2.QtMultimedia` or `PySide6.QtMultimedia` in combination with PySide's `true_property` feature. (#6489)

2.16.31 4.8 (2022-01-06)

Features

- (Windows) Set the executable's build time in PE header to the current time. A custom timestamp can be specified via the `SOURCE_DATE_EPOCH` environment variable to allow reproducible builds. (#6469)
- Add strictly unofficial support for the Termux platform. (#6484)
- Replace the dual-process `onedir` mode on Linux and other Unix-like OSes with a single-process implementation. This makes `onedir` mode on these OSes comparable to Windows and macOS, where single-process `onedir` mode has already been used for a while. (#6407)

Bugfix

- (macOS) Fix regression in generation of `universal2` executables that caused the generated executable to fail `codesign` strict validation. (#6381)
- (Windows) Fix `onefile` extraction behavior when the run-time temporary directory is set to a drive letter. The application's temporary directory is now created directly on the specified drive as opposed to the current directory on the specified drive. (#6051)
- (Windows) Fix compatibility issues with python 3.9.8 from python.org, arising from the lack of embedded manifest in the `python.exe` executable. (#6367)
- (Windows) Fix stack overflow in *pyarmor*-protected frozen applications, caused by the executable's stack being smaller than that of the python interpreter. (#6459)
- (Windows) Fix the `python3.dll` shared library not being found and collected when using Python from MS App Store. (#6390)
- Fix a bug that prevented traceback from uncaught exception to be retrieved and displayed in the windowed bootloader's error reporting facility (uncaught exception dialog on Windows, syslog on macOS). (#6426)
- Fix a crash when a onefile build attempts to overwrite an existing onedir build on macOS or Linux (#6418)
- Fix build errors when a linux shared library (.so) file is collected as a binary on macOS. (#6327)
- Fix build errors when a Windows DLL/PYD file is collected as a binary on a non-Windows OS. (#6327)
- Fix handling of encodings when reading the collected .py source files via `FrozenImporter.get_source()`. (#6143)
- Fix hook loader function not finding hooks if path has whitespaces. (Re-apply the fix that has been inadvertedly undone during the codebase reformatting.) (#6080)
- Windows: Prevent invalid handle errors when an application compiled in *`--windowed`* mode uses `subprocess` without explicitly setting **stdin**, **stdout** and **stderr** to either `PIPE` or `DEVNULL`. (#6364)

Hooks

- (macOS) Add support for Anaconda-installed `PyQtWebEngine`. (#6373)
- Add hooks for `PySide6.QtWebEngineWidgets` and `PyQt6.QtWebEngineWidgets`. The `QtWebEngine` support in PyInstaller requires `Qt6` v6.2.2 or later, so if an earlier version is encountered, we exit with an error instead of producing a defunct build. (#6387)
- Avoid collecting the whole `QtQml` module and its dependencies in cases when it is not necessary (i.e., the application does not use `QtQml` or `QtQuick` modules). The unnecessary collection was triggered due to extension modules being linked against the `libQt5Qml` or `libQt6Qml` shared library, and affected pure widget-based applications (`PySide2` and `PySide6` on Linux) and widget-based applications that use `QtWebEngineWidgets` (`PySide2`, `PySide6`, `PyQt5`, and `PyQt6` on all OSes). (#6447)
- Update `numpy` hook for compatibility with version 1.22; the hook cannot exclude `distutils` and `numpy.distutils` anymore, as they are required by `numpy.testing`, which is used by some external packages, such as `scipy`. (#6474)

Bootloader

- (Windows) Set the bootloader executable's stack size to 2 MB to match the stack size of the python interpreter executable. (#6459)
- Implement single-process `onedir` mode for Linux and Unix-like OSes as a replacement for previously-used two-process implementation. The new mode uses `exec()` without `fork()` to restart the bootloader executable image within the same process after setting up the environment (i.e., the `LD_LIBRARY_PATH` and other environment variables). (#6407)
- Lock the PKG sideload mode in the bootloader unless the executable has a special signature embedded. (#6470)
- When user script terminates with an uncaught exception, ensure that the exception data obtained via `PyErr_Fetch` is normalized by also calling `PyErr_NormalizeException`. Otherwise, trying to format the traceback via `traceback.format_exception` fails in some circumstances, and no traceback can be displayed in the windowed bootloader's error report. (#6426)

Bootloader build

- The bootloader can be force compiled during pip install by setting the environment variable `PYINSTALLER_COMPILE_BOOTLOADER`. (#6384)

2.16.32 4.7 (2021-11-10)

Bugfix

- Fix a bug since v4.6 where certain Unix system directories were incorrectly assumed to exist and resulted in a `FileNotFoundError`. (#6331)

Hooks

- Update `sphinx` hook for compatibility with latest version (4.2.0). (#6330)

Bootloader

- (Windows) Explicitly set `NTDDI_VERSION=0x06010000` and `_WIN32_WINNT=0x0601` when compiling Windows bootloaders to request Windows 7 feature level for Windows headers. The windowed bootloader requires at least Windows Vista feature level, and some toolchains (e.g., mingw cross-compiler on linux) set too low level by default. (#6338)
- (Windows) Remove the check for the unused `windres` utility when compiling with MinGW toolchain. (#6339)
- Replace use of `PyRun_SimpleString` with `PyRun_SimpleStringFlags`. (#6332)

2.16.33 4.6 (2021-10-29)

Features

- Add support for Python 3.10. (#5693)
- (Windows) Embed the manifest into generated `onedir` executables by default, in order to avoid potential issues when user renames the executable (e.g., the manifest not being found anymore due to activation context caching when user renames the executable and attempts to run it before also renaming the manifest file). The old behavior of generating the external manifest file in `onedir` mode can be re-enabled using the `--no-embed-manifest` command-line switch, or via the `embed_manifest=False` argument to `EXE()` in the .spec file. (#6248)
- (Windows) Respect **PEP 239** encoding specifiers in Window's VSVersionInfo files. (#6259)
- Implement basic resource reader for accessing on-filesystem resources (data files) via `importlib.resources` (python >= 3.9) or `importlib_resources` (python <= 3.8). (#5616)
- Ship precompiled wheels for musl-based Linux distributions (such as Alpine or OpenWRT) on `x86_64` and `aarch64`. (#6245)

Bugfix

- (macOS) Ensure that executable pre-processing and post-processing steps (target arch selection, SDK version adjustment, (re)signing) are applied in the stand-alone PKG mode. (#6251)
- (macOS) Robustify the macOS assembly pipeline to work around the issues with the `codesign` utility on macOS 10.13 High Sierra. (#6167)
- (Windows) Fix collection of `sysconfig` platform-specific data module when using MSYS2/MINGW python. (#6118)
- (Windows) Fix displayed script name and exception message in the unhandled exception dialog (windowed mode) when bootloader is compiled using the `MinGW-w64` toolchain. (#6199)
- (Windows) Fix issues in `onedir` frozen applications when the bootloader is compiled using a toolchain that forcibly embeds a default manifest (e.g., the `MinGW-w64` toolchain from `msys2`). The issues range from manifest-related options (e.g., `uac-admin`) not working to windowed frozen application not starting at all (with the `The procedure entry point LoadIconMetric could not be located...` error message). (#6196)
- (Windows) Fix the declared length of strings in the optional embedded product version information resource structure. The declared lengths were twice too long, and resulted in trailing garbage characters when the version information was read using *ctypes* and winver API. (#6219)

- (Windows) Remove the attempt to load the manifest of a `onefile` frozen executable via the activation context, which fails with `An attempt to set the process default activation context failed because the process default activation context was already set.` message that can be observed in debug builds. This approach has been invalid ever since #3746 implemented direct manifest embedding into the `onefile` executable. (#6248)
- (Windows) Suppress missing library warnings for `api-ms-win-core-*` DLLs. (#6201)
- (Windows) Tolerate reading Windows VSVersionInfo files with unicode byte order marks. (#6259)
- Fix `sys.executable` pointing to the external package file instead of the executable when in package side-load mode (`pkg_append=False`). (#6202)
- Fix a runaway glob which caused `ctypes.util.find_library("libfoo")` to non-deterministically pick any library matching `libfoo*` to bundle instead of `libfoo.so`. (#6245)
- Fix compatibility with with MIPS and loongarch64 architectures. (#6306)
- Fix the `FrozenImporter.get_source()` to correctly handle the packages' `__init__.py` source files. This in turn fixes missing-source-file errors for packages that use `pytorch` JIT when the source .py files are collected and available (for example, `kornia`). (#6237)
- Fix the location of the generated stand-alone pkg file when using the side-load mode (`pkg_append=False`) in combination with `onefile` mode. The package file is now placed next to the executable instead of next to the .spec file. (#6202)
- When generating spec files, avoid hard-coding the spec file's location as the `pathex` argument to the `Analysis`. (#6254)

Incompatible Changes

- (Windows) By default, manifest is now embedded into the executable in `onedir` mode. The old behavior of generating the external manifest file can be re-enabled using the `--no-embed-manifest` command-line switch, or via the `embed_manifest=False` argument to `EXE()` in the .spec file. (#6248)

Hooks

- (macOS) Fix compatibility with Anaconda `PyQt5` package. (#6181)
- Add a hook for `pandas.plotting` to restore compatibility with `pandas` 1.3.0 and later. (#5994)
- Add a hook for `QtOpenGLWidgets` for `PyQt6` and `PySide6` to collect the new `QtOpenGLWidgets` module introduced in Qt6 (#6310)
- Add hooks for `QtPositioning` and `QtLocation` modules of the Qt5-based packages (`PySide2` and `PyQt5`) to ensure that corresponding plugins are collected. (#6250)
- Fix compatibility with `PyQt5` 5.9.2 from conda's main channel. (#6114)
- Prevent potential error in hooks for Qt-based packages that could be triggered by a partial `PyQt6` installation. (#6141)
- Update `QtNetwork` hook for `PyQt6` and `PySide6` to collect the new `tls` plugins that were introduced in Qt 6.2. (#6276)
- Update the `gi.repository.GtkSource` hook to accept a module-versions hooksconfig dict in order to allow the hook to be used with GtkSource versions greater than 3.0. (#6267)

Bootloader

- (Windows) Suppress two `snprintf` truncation warnings that prevented bootloader from building with `winlibs MinGW-w64` toolchain. (#6196)
- Update the Linux bootloader cross compiler Dockerfile to allow using the official PyPA base images in place of the dockcross ones. (#6245)

2.16.34 4.5.1 (2021-08-06)

Bugfix

- Fix hook loader function not finding hooks if path has whitespaces. (#6080)

2.16.35 4.5 (2021-08-01)

Features

- (POSIX) Add `exclude_system_libraries` function to the Analysis class for .spec files, to exclude most or all non-Python system libraries from the bundle. Documented in new *POSIX Specific Options* section. (#6022)

Bugfix

- (Cygwin) Add `_MEIPASS` to DLL search path to fix loading of python shared library in onefile builds made in cygwin environment and executed outside of it. (#6000)
- (Linux) Display missing library warnings for "not found" lines in `ldd` output (i.e., `libsomething.so => not found`) instead of quietly ignoring them. (#6015)
- (Linux) Fix spurious missing library warning when `libc.so` points to `ldd`. (#6015)
- (macOS) Fix python shared library detection for non-framework python builds when the library path cannot be inferred from imports of the `python` executable. (#6021)
- (macOS) Fix the crashes in `onedir` bundles of `tkinter`-based applications created using Homebrew python 3.9 and Tcl/Tk 8.6.11. (#6043)
- (macOS) When fixing executable for codesigning, update the value of `vmsize` field in the `__LINKEDIT` segment. (#6039)
- Downgrade messages about missing dynamic link libraries from ERROR to WARNING. (#6015)
- Fix a bytecode parsing bug which caused tuple index errors whilst scanning modules which use `ctypes`. (#6007)
- Fix an error when rhtooks for `pkgutil` and `pkg_resources` are used together. (#6018)
- Fix architecture detection on Apple M1 (#6029)
- Fix crash in windowed bootloader when the traceback for unhandled exception cannot be retrieved. (#6070)
- Improve handling of errors when loading hook entry-points. (#6028)
- Suppress missing library warning for `shiboken2` (`PySide2`) and `shiboken6` (`PySide6`) shared library. (#6015)

Incompatible Changes

- (macOS) Disable processing of Apple events for the purpose of argv emulation in `onedir` application bundles. This functionality was introduced in PyInstaller 4.4 by (#5920) in response to feature requests (#5436) and (#5908), but was discovered to be breaking `tkinter`-based `onedir` bundles made with Homebrew python 3.9 and Tcl/Tk 8.6.11 (#6043). As such, until the cause is investigated and the issue addressed, this feature is reverted/disabled. (#6048)

Hooks

- Add a hook for `pandas.io.formats.style` to deal with indirect import of `jinja2` and the missing template file. (#6010)
- Simplify the `PySide2.QWebEngineWidgets` and `PyQt5.QWebEngineWidgets` by merging most of their code into a common helper function. (#6020)

Documentation

- Add a page describing hook configuration mechanism and the currently implemented options. (#6025)

PyInstaller Core

- Isolate discovery of 3rd-party hook directories into a separate subprocess to avoid importing packages in the main process. (#6032)

Bootloader build

- Allow statically linking zlib on non-Windows specified via either a `--static-zlib` flag or a `PYI_STATIC_ZLIB=1` environment variable. (#6010)

2.16.36 4.4 (2021-07-13)

Features

- (macOS) Implement signing of .app bundle (ad-hoc or with actual signing identity, if provided). (#5581)
- (macOS) Implement support for Apple Silicon M1 (`arm64`) platform and different targets for frozen applications (thin-binary `x86_64`, thin-binary `arm64`, and fat-binary `universal2`), with build-time arch validation and ad-hoc resigning of all collected binaries. (#5581)
- (macOS) In `onedir windowed` (.app bundle) mode, perform an interaction of Apple event processing to convert `odoc` and `GURL` events to `sys.argv` before entering frozen python script. (#5920)
- (macOS) In windowed (.app bundle) mode, always log unhandled exception information to `syslog`, regardless of debug mode. (#5890)
- (Windows) Add support for Python from Microsoft App Store. (#5816)
- (Windows) Implement a custom dialog for displaying information about unhandled exception and its traceback when running in windowed/noconsole mode. (#5890)
- Add **recursive** option to *PyInstaller.utils.hooks.copy_metadata()*. (#5830)

- Add `--codesign-identity` command-line switch to perform code-signing with actual signing identity instead of ad-hoc signing (macOS only). (#5581)
- Add `--osx-entitlements-file` command-line switch that specifies optional entitlements file to be used during code signing of collected binaries (macOS only). (#5581)
- Add `--target-arch` command-line switch to select target architecture for frozen application (macOS only). (#5581)
- Add a splash screen that displays a background image and text: The splash screen can be controlled from within Python using the `pyi_splash` module. A splash screen can be added using the `--splash IMAGE_FILE` option. If optional text is enabled, the splash screen will show the progress of unpacking in onefile mode. This feature is supported only on Windows and Linux. A huge thanks to @Chrisg2000 for programming this feature. (#4354, #4887)
- Add hooks for `PyQt6`. (#5865)
- Add hooks for `PySide6`. (#5865)
- Add option to opt-out from reporting full traceback for unhandled exceptions in windowed mode (Windows and macOS only), via `--disable-windowed-traceback` PyInstaller CLI switch and the corresponding `disable_windowed_traceback` boolean argument to `EXE()` in spec file. (#5890)
- Allow specify which icon set, themes and locales to pack with Gtk applications. Pass a keyword arg `hooksconfig` to Analysis.

```
a = Analysis(["my-gtk-app.py"],
             ...,
             hooksconfig={
                 "gi": {
                     "icons": ["Adwaita"],
                     "themes": ["Adwaita"],
                     "languages": ["en_GB", "zh_CN"]
                 }
             },
             ...)
```

  (#5853)
- Automatically exclude Qt plugins from UPX processing. (#4178)
- Collect distribution metadata automatically. This works by scanning collected Python files for uses of:
  - `pkg_resources.get_distribution()`
  - `pkg_resources.require()`
  - `importlib.metadata.distribution()`
  - `importlib.metadata.metadata()`
  - `importlib.metadata.files()`
  - `importlib.metadata.version()`

  In all cases, the metadata will only be collected if the distribution name is given as a plain string literal. Anything more complex will still require a hook containing *`PyInstaller.utils.hooks.copy_metadata()`*. (#5830)
- Implement support for `pkgutil.iter_modules()`. (#1905)
- Windows: Provide a meaningful error message if given an icon in an unsupported Image format. (#5755)

Bugfix

- (macOS) App bundles built in `onedir` mode now filter out `-psnxxx` command-line argument from `sys.argv`, to keep behavior consistent with bundles built in `onefile` mode. (#5920)
- (macOS) Ensure that the macOS SDK version reported by the frozen application corresponds to the minimum of the SDK version used to build the bootloader and the SDK version used to build the Python library. Having the application report more recent version than Python library and other bundled libraries may result in macOS attempting to enable additional features that are not available in the Python library, which may in turn cause inconsistent behavior and UI issues with `tkinter`. (#5839)
- (macOS) Remove spurious `MacOS/` prefix from `CFBundleExecutable` property in the generated `Info.plist` when building an app bundle. (#4413, #5442)
- (macOS) The drag & drop file paths passed to app bundles built in `onedir` mode are now reflected in `sys.argv`. (#5436)
- (macOS) The file paths passed from the UI (*Open with...*) to app bundles built in `onedir` mode are now reflected in `sys.argv`. (#5908)
- (macOS) Work around the `tkinter` UI issues due to problems with dark mode activation: black `Tk` window with macOS Intel installers from `python.org`, or white text on bright background with Anaconda python. (#5827)
- (Windows) Enable collection of additional VC runtime DLLs (`msvcp140.dll`, `msvcp140_1.dll`, `msvcp140_2.dll`, and `vcruntime140_1.dll`), to allow frozen applications to run on Windows systems that do not have *Visual Studio 2015/2017/2019 Redistributable* installed. (#5770)
- Enable retrieval of code object for `__main__` module via its associated loader (i.e., `FrozenImporter`). (#5897)
- Fix `inspect.getmodule()` failing to resolve module from stack-frame obtained via `inspect.stack()`. (#5963)
- Fix `__main__` module being recognized as built-in instead of module. (#5897)
- Fix a bug in *ctypes dependency scanning* which caused references to be missed if the preceding code contains more than 256 names or 256 literals. (#5830)
- Fix collection of duplicated `_struct` and `zlib` extension modules with mangled filenames. (#5851)
- Fix python library lookup when building with RH SCL python 3.8 or later. (#5749)
- Prevent *`PyInstaller.utils.hooks.copy_metadata()`* from renaming `[...].dist-info` metadata folders to `[...].egg-info` which breaks usage of `pkg_resources.requires()` with *extras*. (#5774)
- Prevent a bootloader executable without an embedded CArchive from being misidentified as having one, which leads to undefined behavior in frozen applications with side-loaded CArchive packages. (#5762)
- Prevent the use of `sys` or `os` as variables in the global namespace in frozen script from affecting the `ctypes` hooks thar are installed during bootstrap. (#5797)
- Windows: Fix EXE being rebuilt when there are no changes. (#5921)

Hooks

- – Add `PostGraphAPI.analysis` attribute. Hooks can access the `Analysis` object through the `hook()` function.
  – Hooks may access a `Analysis.hooksconfig` attribute assigned on `Analysis` construction.

  A helper function *`get_hook_config()`* was defined in `utils.hooks` to get the config. (#5853)
- Add support for `PyQt5` 5.15.4. (#5631)
- Do not exclude `setuptools.py27compat` and `setuptools.py33compat` as they are required by other `setuptools` modules. (#5979)
- Switch the library search order in `ctypes` hooks: first check whether the given name exists as-is, before trying to search for its basename in `sys._MEIPASS` (instead of the other way around). (#5907)

Bootloader

- (macOS) Build bootloader as `universal2` binary by default (can be disabled by passing `--no-universal2` to waf). (#5581)
- Add Tcl/Tk based Splash screen, which is controlled from within Python. The necessary module to create the Splash screen in PyInstaller is under `Splash` available. A huge thanks to @Chrisg2000 for programming this feature. (#4887)
- Provide a Dockerfile to build Linux bootloaders for different architectures. (#5995)

Documentation

- Document the new macOS multi-arch support and code-signing behavior in corresponding sub-sections of `Notes about specific Features`. (#5581)

Bootloader build

- Update `clang` in `linux64` Vagrant VM to `clang-11` from `apt.llvm.org` so it can build `universal2` macOS bootloader. (#5581)
- Update `crossosx` Vagrant VM to build the toolchain from `Command Line Tools for Xcode` instead of full `Xcode package`. (#5581)

2.16.37 4.3 (2021-04-16)

Features

- Provide basic implementation for `FrozenImporter.get_source()` that allows reading source from `.py` files that are collected by hooks as data files. (#5697)
- Raise the maximum allowed size of `CArchive` (and consequently `onefile` executables) from 2 GiB to 4 GiB. (#3939)
- The *unbuffered stdio* mode (the `u` option) now sets the `Py_UnbufferedStdioFlag` flag to enable unbuffered stdio mode in Python library. (#1441)
- Windows: Set EXE checksums. Reduces false-positive detection from antiviral software. (#5579)

- Add new command-line options that map to collect functions from hookutils: `--collect-submodules`, `--collect-data`, `--collect-binaries`, `--collect-all`, and `--copy-metadata`. (#5391)
- Add new hook utility *`collect_entry_point()`* for collecting plugins defined through setuptools entry points. (#5734)

Bugfix

- (macOS) Fix `Bad CPU type in executable` error in helper-spawned python processes when running under `arm64`-only flavor of Python on Apple M1. (#5640)
- (OSX) Suppress missing library error messages for system libraries as those are never collected by PyInstaller and starting with Big Sur, they are hidden by the OS. (#5107)
- (Windows) Change default cache directory to `LOCALAPPDATA` (from the original `APPDATA`). This is to make sure that cached data doesn't get synced with the roaming profile. For this and future versions `AppData\Roaming\pyinstaller` might be safely deleted. (#5537)
- (Windows) Fix `onefile` builds not having manifest embedded when icon is disabled via `--icon NONE`. (#5625)
- (Windows) Fix the frozen program crashing immediately with `Failed to execute script pyiboot01_bootstrap` message when built in `noconsole` mode and with import logging enabled (either via `--debug imports` or `--debug all` command-line switch). (#4213)
- `CArchiveReader` now performs full back-to-front file search for `MAGIC`, allowing `pyi-archive_viewer` to open binaries with extra appended data after embedded package (e.g., digital signature). (#2372)
- Fix `MERGE()` to properly set references to nested resources with their full shared-package-relative path instead of just basename. (#5606)
- Fix `onefile` builds failing to extract files when the full target path exceeds 260 characters. (#5617)
- Fix a crash in `pyi-archive_viewer` when quitting the application or moving up a level. (#5554)
- Fix extraction of nested files in `onefile` builds created in MSYS environments. (#5569)
- Fix installation issues stemming from unicode characters in file paths. (#5678)
- Fix the build-time error under python 3.7 and earlier when `ctypes` is manually added to `hiddenimports`. (#3825)
- Fix the return code if the frozen script fails due to unhandled exception. The return code 1 is used instead of -1, to keep the behavior consistent with that of the python interpreter. (#5480)
- Linux: Fix binary dependency scanner to support changes to ldconfig introduced in `glibc` 2.33. (#5540)
- Prevent `MERGE` (multipackage) from creating self-references for duplicated TOC entries. (#5652)
- PyInstaller-frozen onefile programs are now compatible with `staticx` even if the bootloader is built as position-independent executable (PIE). (#5330)
- Remove dependence on a private function removed in `matplotlib` 3.4.0rc1. (#5568)
- Strip absolute paths from `.pyc` modules collected into `base_library.zip` to enable reproducible builds that are invariant to Python install location. (#5563)
- (OSX) Fix issues with `pycryptodomex` on macOS. (#5583)
- Allow compiled modules to be collected into `base_library.zip`. (#5730)
- Fix a build error triggered by scanning `ctypes.CDLL('libc.so')` on certain Linux C compiler combinations. (#5734)
- Improve performance and reduce stack usage of module scanning. (#5698)

Hooks

- Add support for Conda Forge's distribution of `NumPy`. (#5168)
- Add support for package content listing via `pkg_resources`. The implementation enables querying/listing resources in a frozen package (both PYZ-embedded and on-filesystem, in that order of precedence) via `pkg_resources.resource_exists()`, `resource_isdir()`, and `resource_listdir()`. (#5284)
- Hooks: Import correct typelib for GtkosxApplication. (#5475)
- Prevent `matplotlib` hook from collecting current working directory when it fails to determine the path to matplotlib's data directory. (#5629)
- Update `pandas` hook for compatibility with version 1.2.0 and later. (#5630)
- Update hook for `distutils.sysconfig` to be compatible with pyenv-virtualenv. (#5218)
- Update hook for `sqlalchemy` to support version 1.4.0 and above. (#5679)
- Update hook for `sysconfig` to be compatible with pyenv-virtualenv. (#5018)

Bootloader

- Implement full back-to-front file search for the embedded archive. (#5511)
- Perform file extraction from the embedded archive in a streaming manner in order to limit memory footprint when archive contains large files. (#5551)
- Set the `__file__` attribute in the `__main__` module (entry-point script) to the absolute file name inside the `_MEIPASS`. (#5649)
- Enable cross compiling for FreeBSD from Linux. (#5733)

Documentation

- Doc: Add version spec file option for macOS Bundle. (#5476)
- Update the `Run-time Information` section to reflect the changes in behavior of `__file__` inside the `__main__` module. (#5649)

PyInstaller Core

- Drop support for python 3.5; EOL since September 2020. (#5439)
- Collect python extension modules that correspond to built-ins into `lib-dynload` sub-directory instead of directly into bundle's root directory. This prevents them from shadowing shared libraries with the same basename that are located in a package and loaded via `ctypes` or `cffi`, and also declutters the bundle's root directory. (#5604)

Breaking

- No longer collect `pyconfig.h` and `makefile` for `sysconfig`. Instead of `get_config_h_filename()` and `get_makefile_filename()`, you should use `get_config_vars()` which no longer depends on those files. (#5218)
- The `__file__` attribute in the `__main__` module (entry-point script) is now set to the absolute file name inside the `_MEIPASS` (as if script file existed there) instead of just script filename. This better matches the behavior of `__file__` in the unfrozen script, but might break the existing code that explicitly relies on the old frozen behavior. (#5649)

2.16.38 4.2 (2021-01-13)

Features

- Add hooks utilities to find binary dependencies of Anaconda distributions. (#5213)
- (OSX) Automatically remove the signature from the collected copy of the `Python` shared library, using `codesign --remove-signature`. This accommodates both `onedir` and `onefile` builds with recent python versions for macOS, where invalidated signature on PyInstaller-collected copy of the `Python` library prevents the latter from being loaded. (#5451)
- (Windows) PyInstaller's console or windowed icon is now added at freeze-time and no longer built into the bootloader. Also, using `--icon=NONE` allows to not apply any icon, thereby making the OS to show some default icon. (#4700)
- (Windows) Enable `longPathAware` option in built application's manifest in order to support long file paths on Windows 10 v.1607 and later. (#5424)

Bugfix

- Fix loading of plugin-type modules at run-time of the frozen application: If the plugin path is one character longer than sys._MEIPATH (e.g. "$PWD/p/plugin_1" and "$PWD/dist/main"), the plugin relative-imports a sub-module (of the plugin) and the frozen application contains a module of the same name, the frozen application module was imported. (#4141, #4299)
- Ensure that spec for frozen packages has `submodule_search_locations` set in order to fix compatibility with `importlib_resources` 3.2.0 and later. (#5396)
- Fix: No rebuild if "noarchive" build-option changes. (#5404)
- (OSX) Fix the problem with `Python` shared library collected from recent python versions not being loaded due to invalidated signature. (#5062, #5272, #5434)
- (Windows) PyInstaller's default icon is no longer built into the bootloader, but added at freeze-time. Thus, when specifying an icon, only that icon is contained in the executable and displayed for a shortcut. (#870, #2995)
- (Windows) Fix "toc is bad" error messages when passing a `VSVersionInfo` as the `version` parameter to `EXE()` in a `.spec` file. (#5445)
- (Windows) Fix exception when trying to read a manifest from an exe or dll. (#5403)
- (Windows) Fix the `--runtime-tmpdir` option by creating paths if they don't exist and expanding environment variables (e.g. `%LOCALAPPDATA%`). (#3301, #4579, #4720)

Hooks

- (GNU/Linux) Collect `xcbglintegrations` and `egldeviceintegrations` plugins as part of `Qt5Gui`. (#5349)
- (macOS) Fix: Unable to code sign apps built with GTK (#5435)
- (Windows) Add a hook for `win32ctypes.core`. (#5250)
- Add hook for `scipy.spatial.transform.rotation` to fix compatibility with SciPy 1.6.0. (#5456)
- Add hook-gi.repository.GtkosxApplication to fix TypeError with Gtk macOS apps. (#5385)
- Add hooks utilities to find binary dependencies of Anaconda distributions. (#5213)
- Fix the `Qt5` library availability check in `PyQt5` and `PySide2` hooks to re-enable support for `Qt5` older than 5.8. (#5425)
- Implement `exec_statement_rc()` and `exec_script_rc()` as exit-code returning counterparts of `exec_statement()` and `exec_script()`. Implement `can_import_module()` helper for hooks that need to query module availability. (#5301)
- Limit the impact of a failed sub-package import on the result of `collect_submodules()` to ensure that modules from all other sub-packages are collected. (#5426)
- Removed obsolete `pygame` hook. (#5362)
- Update `keyring` hook to collect metadata, which is required for backend discovery. (#5245)

Bootloader

- (GNU/Linux) Reintroduce executable resolution via `readlink()` on `/proc/self/exe` and preserve the process name using `prctl()` with `PR_GET_NAME` and `PR_SET_NAME`. (#5232)
- (Windows) Create temporary directories with user's SID instead of `S-1-3-4`, to work around the lack of support for the latter in `wine`. This enables `onefile` builds to run under `wine` again. (#5216)
- (Windows) Fix a bug in path-handling code with paths exceeding `PATH_MAX`, which is caused by use of `_snprintf` instead of `snprintf` when building with MSC. Requires Visual Studio 2015 or later. Clean up the MSC codepath to address other compiler warnings. (#5320)
- (Windows) Fix building of bootloader's test suite under Windows with Visual Studio. This fixes build errors when `cmocka` is present in the build environment. (#5318)
- (Windows) Fix compiler warnings produced by MinGW 10.2 in order to allow building the bootloader without having to suppress the warnings. (#5322)
- (Windows) Fix `windowed+debug` bootloader variant not properly displaying the exception message and traceback information when the frozen script terminates due to uncaught exception. (#5446)

PyInstaller Core

- (Windows) Avoid using UPX with DLLs that have control flow guard (CFG) enabled. (#5382)
- Avoid using `.pyo` module file suffix (removed since PEP-488) in `noarchive` mode. (#5383)
- Improve support for `PEP-420` namespace packages. (#5354)
- Strip absolute paths from `.pyc` modules collected in the CArchive (PKG). This enables build reproducibility without having to match the location of the build environment. (#5380)

2.16.39 4.1 (2020-11-18)

Features

- Add support for Python 3.9. (#5289)
- Add support for Python 3.8. (#4311)

Bugfix

- Fix endless recursion if a package's `__init__` module is an extension module. (#5157)
- Remove duplicate logging messages (#5277)
- Fix sw_64 architecture support (#5296)
- (AIX) Include python-malloc labeled libraries in search for libpython. (#4210)

Hooks

- Add `exclude_datas`, `include_datas`, and `filter_submodules` to `collect_all()`. These arguments map to the `excludes` and `includes` arguments of `collect_data_files`, and to the *filter* argument of `collect_submodules`. (#5113)
- Add hook for difflib to not pull in doctests, which is only required when run as main program.
- Add hook for distutils.util to not pull in lib2to3 unittests, which will be rearly used in frozen packages.
- Add hook for heapq to not pull in doctests, which is only required when run as main program.
- Add hook for multiprocessing.util to not pull in python test-suite and thus e.g. tkinter.
- Add hook for numpy._pytesttester to not pull in pytest.
- Add hook for pickle to not pull in doctests and argpargs, which are only required when run as main program.
- Add hook for PIL.ImageFilter to not pull numpy, which is an optional component.
- Add hook for setuptools to not pull in numpy, which is only imported if installed, not mean to be a dependency
- Add hook for zope.interface to not pull in pytest unittests, which will be rearly used in frozen packages.
- Add hook-gi.repository.HarfBuzz to fix Typelib error with Gtk apps. (#5133)
- Enable overriding Django settings path by *DJANGO_SETTINGS_MODULE* environment variable. (#5267)
- Fix *collect_system_data_files* to scan the given input path instead of its parent. File paths returned by *collect_all_system_data* are now relative to the input path. (#5110)
- Fix argument order in `exec_script()` and `eval_script()`. (#5300)
- Gevent hook does not unnecessarily bundle HTML documentation, __pycache__ folders, tests nor generated .c and .h files (#4857)
- gevent: Do not pull in test-suite (still to be refined)
- Modify hook for `gevent` to exclude test submodules. (#5201)
- Prevent .pyo files from being collected by collect_data_files when include_py_files is False. (#5141)
- Prevent output to `stdout` during module imports from ending up in the modules list collected by `collect_submodules`. (#5244)

- Remove runtime hook and fix regular hook for matplotlib's data to support `matplotlib>=3.3.0`, fix deprecation warning on version 3.1<= & <3.3, and behave normally for versions <3.1. (#5006)
- Remove support for deprecated PyQt4 and PySide (#5118, #5126)
- setuptools: Exclude outdated compat modules.
- Update `sqlalchemy` hook to support v1.3.19 and later, by adding `sqlalchemy.ext.baked` as a hidden import (#5128)
- Update `tkinter` hook to collect Tcl modules directory (`tcl8`) in addition to Tcl/Tk data directories. (#5175)
- (GNU/Linux) {PyQt5,PySide2}.QtWebEngineWidgets: fix search for extra NSS libraries to prevent an error on systems where /lib64/nss/*.so comes up empty. (#5149)
- (OSX) Avoid collecting data from system Tcl/Tk framework in `tkinter` hook as we do not collect their shared libraries, either. Affects only python versions that still use the system Tcl/Tk 8.5. (#5217)
- (OSX) Correctly locate the tcl/tk framework bundled with official python.org python builds from v.3.6.5 on. (#5013)
- (OSX) Fix the QTWEBENGINEPROCESS_PATH set in PyQt5.QtWebEngineWidgets rthook. (#5183)
- (OSX) PySide2.QtWebEngineWidgets: add QtQmlModels to included libraries. (#5150)
- (Windows) Remove the obsolete python2.4-era `_handle_broken_tcl_tk` work-around for old virtual environments from the `tkinter` hook. (#5222)

Bootloader

- Fix freeing memory allocated by Python using `free()` instead of `PyMem_RawFree()`. (#4441)
- (GNU/Linux) Avoid segfault when temp path is missing. (#5255)
- (GNU/Linux) Replace a `strncpy()` call in `pyi_path_dirname()` with `snprintf()` to ensure that the resulting string is always null-terminated. (#5212)
- (OSX) Added capability for already-running apps to accept URL & drag'n drop events via Apple Event forwarding (#5276)
- (OSX) Bump `MACOSX_DEPLOYMENT_TARGET` from 10.7 to 10.13. (#4627, #4886)
- (OSX) Fix to reactivate running app on "reopen" (#5295)
- (Windows) Use `_wfullpath()` instead of `_fullpath()` in `pyi_path_fullpath` to allow non-ASCII characters in the path. (#5189)

Documentation

- Add zlib to build the requirements in the Building the Bootlooder section of the docs. (#5130)

PyInstaller Core

- Add informative message what do to if RecurrsionError occurs. (#4406, #5156)
- Prevent a local directory with clashing name from shadowing a system library. (#5182)
- Use module loaders to get module content instea of an quirky way semming from early Python 2.x times. (#5157)
- (OSX) Exempt the `Tcl/Tk` dynamic libraries in the system framework from relative path overwrite. Fix missing `Tcl/Tk` dynlib on older python.org builds that still make use of the system framework. (#5172)

Test-suite and Continuous Integration

- Replace `skipif_xxx` for platform-specific tests by markers. (#1427)
- Test/CI: Test failures are automatically retried once. (#5214)

Bootloader build

- Fix AppImage builds that were broken since PyInstaller 3.6. (#4693)
- Update build system to use Python 3.
- OSX: Fixed the ineffectiveness of the `--distpath` argument for the `BUNDLE` step. (#4892)
- OSX: Improve codesigning and notarization robustness. (#3550, #5112)
- OSX: Use high resolution mode by default for GUI applications. (#4337)

2.16.40 4.0 (2020-08-08)

Features

- Provide setuptools entrypoints to enable other packages to provide PyInstaller hooks specific to that package, along with tests for these hooks.

  Maintainers of Python packages requiring hooks are invited to use this new feature and provide up-to-date PyInstaller support along with their package. This is quite easy, see our sample project for more information (#4232, #4301, #4582). Many thanks to Bryan A. Jones for implementing the important parts.
- A new package pyinstaller-hooks-contrib provides monthly updated hooks now. This package is installed automatically when installing PyInstaller, but can be updated independently. Many thanks to Legorooj for setting up the new package and moving the hooks there.
- Added the `excludes` and `includes` arguments to the hook utility function `collect_data_files`.
- Change the hook collection order so that the hook-priority is command line, then entry-point, then PyInstaller builtins. (#4876)

Bugfix

- (AIX) Include python-malloc labeled libraries in search for libpython. (#4738)
- (win32) Fix Security Alerts caused by subtle implementation differences between posix anf windows in `os.path.dirname()`. (#4707)
- (win32) Fix struct format strings for versioninfo. (#4861)
- (Windows) cv2: bundle the *opencv_videoio_ffmpeg*.dll*, if available. (#4999)
- (Windows) GLib: bundle the spawn helper executables for *g_spawn** API. (#5000)
- (Windows) PySide2.QtNetwork: search for SSL DLLs in *PrefixPath* in addition to *BinariesPath*. (#4998)
- (Windows) When building with 32-bit python in onefile mode, set the `requestedExecutionLevel` manifest key every time and embed the manifest. (#4992)
- – (AIX) Fix uninitialized variable. (#4728, #4734)
- Allow building on a different drive than the source. (#4820)
- Consider Python<version> as possible library binary path. Fixes issue where python is not found if Python3 is installed via brew on OSX (#4895)
- Ensure shared dependencies from onefile packages can be opened in the bootloader.
- Ensuring repeatable builds of base_library.zip. (#4654)
- Fix `FileNotFoundError` showing up in `utils/misc.py` which occurs when a namespace was processed as an filename. (#4034)
- Fix multipackaging. The *MERGE* class will now have the correct relative paths between shared dependencies which can correctly be opened by the bootloader. (#1527, #4303)
- Fix regression when trying to avoid hard-coded paths in .spec files.
- Fix SIGTSTP signal handling to allow typing Ctrl-Z from terminal. (#4244)
- Update the base library to support encrypting Python bytecode (`--key` option) again. Many thanks to Matteo Bertini for finally fixing this. (#2365, #3093, #3133, #3160, #3198, #3316, #3619, #4241, #4652)
- When stripping the leading parts of paths in compiled code objects, the longest possible import path will now be stripped. (#4922)

Incompatible Changes

- Remove support for Python 2.7. The minimum required version is now Python 3.5. The last version supporting Python 2.7 was PyInstaller 3.6. (#4623)
- Many hooks are now part of the new *pyinstaller-hooks-contrib* repository. See below for a detailed list.

Hooks

- Add hook for `scipy.stats._stats` (needed for scipy since 1.5.0). (#4981)
- Prevent hook-nltk from adding non-existing directories. (#3900)
- Fix `importlib_resources` hook for modern versions (after 1.1.0). (#4889)
- Fix hidden imports in pkg_resources and packaging (#5044)
  - Add yet more hidden imports to pkg_resources hook.
  - Mirror the pkg_resources hook for packaging which may or may not be duplicate of `pkg_resources._vendor.packaging`.
- Update pkg_resources hook for setuptools v45.0.0.
- Add QtQmlModels to included libraries for QtWebEngine on OS X (#4631).
- Fix detecting Qt5 libraries and dependencies from conda-forge builds (#4636).
- Add an AssertionError message so that users who get an error due to Hook conflicts can resolve it (#4626).
- These hooks have been moved to the new pyinstaller-hooks-contrib repository: BTrees, Crypto, Cryptodome, IPython, OpenGL, OpenGL_accelerate, Xlib, accessible_output2, adios, aliyunsdkcore, amazonproduct, appdirs, appy, astor, astroid, astropy, avro, bacon, boto, boto3, botocore, certifi, clr, countrycode, cryptography, cv2, cx_Oracle, cytoolz, dateparser, dclab, distorm3, dns, docutils, docx, dynaconf, enchant, enzyme, eth_abi, eth_account, eth_hash, eth_keyfile, eth_utils, faker, flex, fmpy, gadfly, gooey, google.*, gst, gtk, h5py, httplib, httplib2, imageio, imageio_ffmpeg, jedi, jinja2, jira, jsonpath_rw_ext, jsonschema, jupyterlab, kinterbasdb, langcodes, lensfunpy, libaudioverse, llvmlite, logilab, lxml, lz4, magic, mako, markdown, migrate, mpl_toolkits, mssql, mysql, nacl, names, nanite, nbconvert, nbdime, nbformat, ncclient, netCDF4, nltk, nnpy, notebook, numba, openpyxl, osgeo, passlib, paste, patsy, pendulum, phonenumbers, pint, pinyin, psychopy, psycopg2, pubsub, pyarrow, pycountry, pycparser, pyexcel, pyexcelerate, pylint, pymssql, pyodbc, pyopencl, pyproj, pysnmp, pytest, pythoncom, pyttsx, pywintypes, pywt, radicale, raven, rawpy, rdflib, redmine, regex, reportlab, reportlab, resampy, selenium, shapely, skimage, sklearn, sound_lib, sounddevice, soundfile, speech_recognition, storm, tables, tcod, tensorflow, tensorflow_corethon, text_unidecode, textdistance, torch, ttkthemes, ttkwidgets, u1db, umap, unidecode, uniseg, usb, uvloop, vtkpython, wavefile, weasyprint, web3, webrtcvad, webview, win32com, wx, xml.dom, xml.sax, xsge_gui, zeep, zmq.
- These hooks have been added while now moved to the new *pyinstaller-hooks-contrib* repository: astor (#4400, #4704), argon2 (#4625) bcrypt. (#4735), (Bluetooth Low Energy platform Agnostic Klient for Python) (#4649) jaraco.text (#4576, #4632), LightGBM. (#4634), xmldiff (#4680), puremagic (identify a file based off it's magic numbers) (#4709) webassets (#4760), tensorflow_core (to support tensorflow module forwarding logic (#4400, #4704)
- These changes have been applied to hooks now moved to the new *pyinstaller-hooks-contrib* repository
  - Update Bokeh hook for v2.0.0. (#4742, #4746)
  - Fix shapely hook on Windows for non-conda shapely installations. (#2834, #4749)

Bootloader

- Rework bootloader from using strcpy/strncpy with "is this string terminated"-check to use snprintf(); check success at more places. (This started from fixing GCC warnings for strncpy and strncat.)
- Fix: When copying files, too much data was copied in most cases. This corrupted the file and inhibited using shared dependencies. (#4303)
- In debug and windowed mode, show the traceback in dialogs to help debug pyiboot01_bootstrap errors. (#4213, #4592)
- Started a small test-suite for bootloader basic functions. (#4585)

Documentation

- Add platform-specific usage notes and bootloader build notes for AIX. (#4731)

PyInstaller Core

- Provide setuptools entrypoints to enable other packages to provide PyInstaller hooks specific to that package, along with tests for these hooks. See https://github.com/pyinstaller/hooksample for more information. (#4232, #4582)

Bootloader build

- (AIX) The argument -X32 or -X64 is not recognized by the AIX loader - so this code needs to be removed. (#4730, #4731)
- (OSX) Allow end users to override MACOSX_DEPLOYMENT_TARGET and mmacosx-version-min via environment variables and set 10.7 as the fallback value for both. (#4677)
- Do not print info about `--noconfirm` when option is already being used. (#4727)
- Update **`waf`** to version 2.0.20 (#4839)

2.16.41 Older Versions

Changelog for PyInstaller 3.0 – 3.6

3.6 (2020-01-09)

Important: This is the last release of PyInstaller supporting Python 2.7. Python 2 is end-of-life, many packages are about to drop support for Python 2.7 - or already did it.

Security

- [SECURITY] (Win32) Fix CVE-2019-16784: Local Privilege Escalation caused by insecure directory permissions of sys._MEIPATH. This security fix effects all Windows software frozen by PyInstaller in "onefile" mode. While PyInstaller itself was not vulnerable, all Windows software frozen by PyInstaller in "onefile" mode is vulnerable.

  If you are using PyInstaller to freeze Windows software using "onefile" mode, you should upgrade PyInstaller and rebuild your software.

Features

- (Windows): Applications built in windowed mode have their debug messages sent to any attached debugger or DebugView instead of message boxes. (#4288)
- Better error message when file exists at path we want to be dir. (#4591)

Bugfix

- (Windows) Allow usage of *VSVersionInfo* as version argument to EXE again. (#4381, #4539)
- (Windows) Fix MSYS2 dll's are not found by modulegraph. (#4125, #4417)
- (Windows) The temporary copy of bootloader used add resources, icons, etc. is not created in –workpath instead of in %TEMP%. This fixes issues on systems where the anti-virus cleans %TEMP% immediately. (#3869)
- Do not fail the build when `ldconfig` is missing/inoperable. (#4261)
- Fixed loading of IPython extensions. (#4271)
- Fixed pre-find-module-path hook for *distutils* to be compatible with *virtualenv >= 16.3*. (#4064, #4372)
- Improve error reporting when the Python library can't be found. (#4162)

Hooks

- Add hook for avro (serialization and RPC framework) (#4388), django-babel (#4516), enzyme (#4338), google.api (resp. google.api.core) (#3251), google.cloud.bigquery (#4083, #4084), google.cloud.pubsub (#4446), google.cloud.speech (#3888), nnpy (#4483), passlib (#4520), pyarrow (#3720, #4517), pyexcel and its plugins io, ods, ods3, odsr, xls, xlsx, xlsxw (#4305), pysnmp (#4287), scrapy (#4514), skimage.io (#3934), sklearn.mixture (#4612), sounddevice on macOS and Windows (#4498), text-unidecode (#4327, #4530), the google-cloud-kms client library (#4408), ttkwidgets (#4484), and webrtcvad (#4490).
- Correct the location of Qt translation files. (#4429)
- Exclude imports for pkg_resources to fix bundling issue. (#4263, #4360)
- Fix hook for pywebview to collect all required libraries and data-files. (#4312)
- Fix hook numpy and hook scipy to account for differences in location of extra dlls on Windows. (#4593)
- Fix pysoundfile hook to bundle files correctly on both OSX and Windows. (#4325)
- Fixed hook for pint to also copy metadata as required to retrieve the version at runtime. (#4280)
- Fixed PySide2.QtNetwork hook by mirroring PyQt5 approach. (#4467, #4468)
- Hook for pywebview now collects data files and dynamic libraries only for the correct OS (Windows). Hook for pywebview now bundles only the required 'lib' subdirectory. (#4375)

- Update hooks related to PySide2.QtWebEngineWidgets, ensure the relevant supporting files required for a QtWebEngineView are copied into the distribution. (#4377)
- Update PyQt5 loader to support PyQt >=5.12.3. (#4293, #4332)
- Update PyQt5 to package 64-bit SSL support DLLs. (#4321)
- Update PyQt5 to place OpenGL DLLs correctly for PyQt >= 5.12.3. (#4322)
- (GNU/Linux) Make hook for GdkPixbuf compatible with Ubuntu and Debian (#4486).

Bootloader

- (OSX): Added support for appending URL to program arguments when applications is launched from custom protocol handler. (#4397, #4399)
- (POSIX) For one-file binaries, if the program is started via a symlink, the second process now keeps the basename of the symlink. (#3823, #3829)
- (Windows) If bundled with the application, proactivley load `ucrtbase.dll` before loading the Python library. This works around unresolved symbol errors when loading `python35.dll` (or later) on legacy Windows (7, 8, 8.1) systems with Universal CRT update is not installed. (#1566, #2170, #4230)
- Add our own implementation for `strndup` and `strnlen` to be used on platforms one of these is missing.

PyInstaller Core

- Now uses hash based *.pyc* files as specified in **PEP 552** in *base_library.zip* when using Python 3.7 (#4096)

Bootloader build

- (MinGW-w64) Fix .rc.o file not found error. (#4501, #4586)
- Add a check whether `strndup` and `strnlen` are available.
- Added OpenBSD support. (#4545)
- Fix build on Solaris 10.
- Fix checking for compiler flags in *configure* phase. The check for compiler flags actually did never work. (#4278)
- Update url for public key in update-waf script. (#4584)
- Update waf to version 2.0.19.

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-live.
- 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), gooey (#3773), GtkSourceView (#3893), imageio_ffmpeg (#4051), importlib_metadata and importlib_resources (#4095), jsonpath_rw_ext (#3841), jupyterlab (#3951), lz4 (#3710), magic (#4267), nanite (#3860), nbconvert (#3947), nbdime (#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), rdflib (#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.`
- 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.

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 `QtQml.QQmlEngine`.
  - 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 bootloder 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 by 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 imports `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 a 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)

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*.

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 (#2066), pylint, pymssql, pyopencl, pyproj (#2677), pytest (#2119), qtawesome (#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 PyGObject 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

- Add *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 mainteriner 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-save? (#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.%dm.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.

3.2.1 (2017-01-15)

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

Fixed the following issues:

- Don't fail if working directory already exists (#1994)
- Avoid encoding errors in main script (#1976)
- Fix hasher digest bytes not str (#2229, #2230)
- (Windows) Fix additional dependency on the msvcrt10.dll (#1974)

- (Windows) Correctly decode a bytes object produced by pefile (#1981)
- (Windows) Package `pefile` with pyinstaller. This partially undoes some changes in 3.2 in which the packaged pefiles were removed to use the pypi version instead. The pypi version was considerably slower in some applications, and still has a couple of small issues on PY3. (#1920)
- (OS X) PyQt5 packaging issues on MacOS (#1874)
- (OS X) Replace run-time search path keyword (#1965)
- (OS X) (Re-) add argv emulation for OSX, 64-bit (#2219)
- (OS X) use decode("utf-8") to convert bytes in getImports_macholib() (#1973)
- (Bootloader) fix segfaults (#2176)
- (setup.py) pass option –no-lsb on GNU/Linux only (#1975)
- Updates and fixes in documentation, manuals, et al. (#1986, 2002, #2153, #2227, #2231)

3.2 (2016-05-03)

- Even the "main" script is now byte-compiled (#1847, #1856)
- The manual is on readthedocs.io now (#1578)
- On installation try to compile the bootloader if there is none for the current platform (#1377)
- (Unix) Use `objcopy` to create a valid ELF file (#1812, #1831)
- (Linux): Compile with `_FORTIFY_SOURCE` (#1820)
- New, updated and fixed hooks: CherryPy (#1860), Cryptography (#1425, #1861), enchant (1562), gi.repository.GdkPixbuf (#1843), gst (#1963), Lib2to3 (#1768), PyQt4, PyQt5, PySide (#1783, #1897, #1887), SciPy (#1908, #1909), sphinx (#1911, #1912), sqlalchemy (#1951), traitlets wx.lib.pubsub (#1837, #1838),
- For windowed mode add `isatty()` for our dummy NullWriter (#1883)
- Suppress "Failed to execute script" in case of SystemExit (#1869)
- Do not apply Upx compressor for bootloader files (#1863)
- Fix absolute path for lib used via ctypes (#1934)
- (OSX) Fix binary cache on NFS (#1573, #1849)
- (Windows) Fix message in grab_version (#1923)
- (Windows) Fix wrong icon 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 bytes-code when scanning for ctypes (#1620).
- Fix PyCrypto modules to work with crypto feature (`--key` option) (#1663).
- Fix problems with `excludedimports` in some hook excluding the named modules even if used elswhere (#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, PyGobject, pymssql, PyNaCl, PySiDe.QtCore, PySide.QtGui, rawpy, requests, scapy, scipy, six, SpeechRecognition, u1db, weasyprint, Xlib.
- Hook fixes: babel, ctypes, django, IPython, pint, PyEnchant, Pygments, PyQt5, PySide, pyusb, sphinx, sqlalchemy, tkinter, wxPython.
- Add support for automatically including data files from eggs.
- Add support for directory eggs support.
- Add support for all kind of namespace packages e.g. `zope.interface`, PEP302 (#502, #615, #665, #1346).
- Add support for `pkgutil.extend_path()`.
- New option `--key` to obfuscate the Python bytecode.

- New option `--exclude-module` to ignore a specific module or package.
- (Windows) New option `--uac-admin` to request admin permissions before starting the app.
- (Windows) New option `--uac-uiaccess` allows an elevated application to work with Remote Desktop.
- (Windows) New options for Side-by-side Assembly searching:
  - `--win-private-assemblies` bundled Shared Assemblies into the application will be changed into Private Assemblies
  - `--win-no-prefer-redirects` while searching for Assemblies PyInstaller will prefer not to follow policies that redirect to newer versions.
- (OSX) New option `--osx-bundle-identifier` to set .app bundle identifier.
- (Windows) Remove old COM server support.
- Allow override PyInstaller default config directory by environment variable `PYINSTALLER_CONFIG_DIR`.
- Add FreeBSD support.
- AIX fixes.
- Solaris fixes.
- Use library modulegraph for module dependency analysis.
- Bootloader debug messages `LOADER: ...` printed to stderr.
- PyInstaller no longer extends `sys.path` and bundled 3rd-party libraries do not interfere with their other versions.
- Enhancemants 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 useful for some specialized preprocessing just for the frozen executable. E.g. this option can be used to set SIP api v2 for PyQt4.
- Fix runtime option –Wignore.
- Rename utils to lowercase: 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 executales. Native Python implementation is kept unchanged.
- Drop old crypto code. This feature was never completed.
- Drop bootloader dependency on Python headers for compilation.

- (Windows) Recompile bootloaders with VS2008 to ensure win2k compatibility.
- (Windows) Use 8.3 filenames for homepath/temppath.
- Add prefix LOADER to the debug text from bootloader.
- Allow running PyInstaller programmatically.
- Move/Rename some files, code refactoring.
- Add more tests.
- Tilde is in PyInstaller recognized as $HOME variable.

2.0 (2012-08-08)

- Minimum supported Python version is 2.3.
- (OSX) Add support for Mac OS X 64-bit
- (OSX) Add support Mac OS X 10.7 (Lion) and 10.8 (Mountain Lion).
- (OSX) With argument –windowed PyInstaller creates application bundle (.app)
- automatically.
- Add experimental support for AIX (thanks to Martin Gamwell Dawids).
- Add experimental support for Solaris (thanks to Hywel Richards).
- Add Multipackage function to create a collection of packages to avoid
- library duplication. See documentation for more details.
- New symplified command line interface. Configure.py/Makespec.py/Build.py
- replaced by pyinstaller.py. See documentation for more details.
- Removed cross-building/bundling feature which was never really finished.
- Added option –log-level to all scripts to adjust level of output (thanks to Hartmut Goebel).
- rthooks.dat moved to support/rthooks.dat
- Packaged executable now returns the same return-code as the
- unpackaged script (thanks to Brandyn White).
- Add import hook for PyUSB (thanks to Chien-An “Zero” Cho).
- Add import hook for wx.lib.pubsub (thanks to Daniel Hyams).
- Add import hook for pyttsx.
- Improve import hook for Tkinter.
- Improve import hook for PyQt4.
- Improve import hook for win32com.
- Improve support for running PyInstaller in virtualenv.
- Add cli options –additional-hooks-dir and –hidden-import.
- Remove cli options -X, -K, -C, –upx, –tk, –configfile, –skip-configure.
- UPX is used by default if available in the PATH variable.
- Remove compatibility code for old platforms (dos, os2, MacOS 9).

- Use Python logging system for message output (thanks to Hartmut Goebel).
- Environment variable MEIPASS2 is accessible as sys._MEIPASS.
- Bootloader now overrides PYTHONHOME and PYTHONPATH. PYTHONHOME and PYTHONPATH is set to the value of MEIPASS2 variable.
- Bootloader uses absolute paths.
- (OSX) Drop dependency on otool from Xcode on Mac OSX.
- (OSX) Fix missing qt_menu.nib in dist directory when using PyQt4.
- (OSX) Bootloader does not use DYLD_LIBRARY_PATH on Mac OS X anymore. @loader_path is used instead.
- (OSX) Add support to detect .dylib dependencies on Mac OS X containing @executable_path, @loader_path and @rpath.
- (OSX) Use macholib to detect dependencies on dynamic libraries.
- Improve test suite.
- Improve source code structure.
- Replace os.system() calls by suprocess 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.
- 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.
- Preliminar 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)
  - cElementTree (both native version and Python 2.5 version)
  - lxml
  - SQLAlchemy (thanks to Greg Copeland)
  - email in Python 2.5 (though it does not support the old-style Python 2.4 syntax with Python 2.5)
  - gadfly
  - PyQWt5
  - mako
  - Improved PyGTK (thanks to Marco Bonifazi and foxx).
  - paste (thanks to Jamie Kirkpatrick)
  - matplotlib
- Add fix for the very annoying "MSVCRT71 could not be extracted" bug, which was caused by the DLL being packaged twice (thanks to Idris Aykun).
- Removed C++-style comments from the bootloader for compatibility with the AIX compiler.
- Fix support for .py files with DOS line endings under Linux (fixes PyOpenGL).
- Fix support for PIL when imported without top-level package ("import Image").
- Fix PyXML import hook under NT (thanks to Lorenzo Mancini)
- Fixed problem with PyInstaller picking up the wrong copy of optparse.
- Improve correctness of the binary cache of UPX'd/strip'd files. This fixes problems when switching between multiple versions of the same third-party library (like e.g. wxPython allows to do).
- Fix a stupid bug with modules importing optparse (under Linux) (thanks to Louai Al-Khanji).

- Under Python 2.4+, if an exception is raised while importing a module inside a package, the module is now removed from the parent's namespace (to match the behaviour of Python itself).
- Fix random race-condition at startup of one-file packages, that was causing this exception to be generated: "PYZ entry 'encodings' (0j) is not a valid code object".
- Fix problem when having unicode strings among path elements.
- Fix random exception ("bad file descriptor") with "prints" in non-console mode (actually a pythonw "bug" that's fixed in Python 3.0).
- Sometimes the temporary directory did not get removed upon program exit, when running on Linux.
- Fixed random segfaults at startup on 64-bit platforms (like x86-64).

1.3 (2006-12-20)

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

1.2 (2006-06-29)

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

1.1 (2006-02-13)

- (Windows) Make single-file packages not depend on MSVCRT71.DLL anymore, even under Python 2.4. You can eventually ship your programs really as single-file executables, even when using the newest Python version!
- Fix problem with incorrect python path detection. Now using helpers from distutils.
- Fix problem with rare encodings introduced in newer Python versions: now all the encodings are automatically found and included, so this problem should be gone forever.
- Fix building of COM servers (was broken in 1.0 because of the new build system).
- Mimic Python 2.4 behaviour with broken imports: sys.modules is cleaned up afterwise. This allows to package 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)

2.17 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:

2.17.1 Contributions to PyInstaller 6.6.0

- Rok Mandeljc
- Brénainn Woodsend
- Olliver Aikenhead
- RoboSchmied

2.17.2 Contributions to PyInstaller 6.5.0

- Rok Mandeljc
- Brénainn Woodsend
- Amir Rossert
- Geo5

2.17.3 Contributions to PyInstaller 6.4.0

- Rok Mandeljc
- Dan Yeaw
- Andrey Marakulin
- mbushkov
- xuanzhi33

2.17.4 Contributions to PyInstaller 6.3.0

- Rok Mandeljc
- Dan Yeaw

2.17.5 Contributions to PyInstaller 6.2.0

- Rok Mandeljc
- David Baumgold

2.17.6 Contributions to PyInstaller 6.1.0

- Rok Mandeljc
- Sebastian Thomschke

2.17.7 Contributions to PyInstaller 6.0.0

- Rok Mandeljc
- Brénainn Woodsend
- Benedikt Würkner
- Blank
- Brandon
- James Gerity
- Lorenzo Villani
- axoroll7
- byehack

- coolcatco888
- gentlegiantJGC

2.17.8 Contributions to PyInstaller 5.13.2

- Rok Mandeljc

2.17.9 Contributions to PyInstaller 5.13.1

- Rok Mandeljc
- Brénainn Woodsend
- James Gerity
- Lorenzo Villani
- coolcatco888
- gentlegiantJGC

2.17.10 Contributions to PyInstaller 5.13.0

- Dan Yeaw
- Rok Mandeljc
- Brénainn Woodsend
- Fabian Dröge
- Levin Ma

2.17.11 Contributions to PyInstaller 5.12.0

- Rok Mandeljc
- Brénainn Woodsend
- Joshua Bronson
- caption

2.17.12 Contributions to PyInstaller 5.11.0

- Rok Mandeljc
- cat (also known as *0xb8*)
- eduardomotta-emottasistemas

2.17.13 Contributions to PyInstaller 5.10.1

- Rok Mandeljc
- Christian Clauss

2.17.14 Contributions to PyInstaller 5.10.0

- Rok Mandeljc
- Michael Shigorin
- V. Armando Solé

2.17.15 Contributions to PyInstaller 5.9.0

- Brénainn Woodsend
- Hugo van Kemenade
- Rok Mandeljc
- Ievgen Popovych

2.17.16 Contributions to PyInstaller 5.8.0

- Rok Mandeljc
- Brénainn Woodsend
- Arjan Molenaar
- Breeze
- Ievgen Popovych
- João Vitor
- bersbersbers

2.17.17 Contributions to PyInstaller 5.7.0

- Rok Mandeljc
- Brénainn Woodsend
- Dan Yeaw
- Rumbelows
- Shoshana Berleant

2.17.18 Contributions to PyInstaller 5.6.2

- Rok Mandeljc
- bersbersbers

2.17.19 Contributions to PyInstaller 5.6.1

- Timmy Welch
- Rok Mandeljc
- Brénainn Woodsend

2.17.20 Contributions to PyInstaller 5.6

- Rok Mandeljc
- Brénainn Woodsend
- Padsala Tushal

2.17.21 Contributions to PyInstaller 5.5

- Rok Mandeljc
- Jasper Harrison
- Alex
- Andreas Schwab
- jsagarribay

2.17.22 Contributions to PyInstaller 5.4.1

- Rok Mandeljc

2.17.23 Contributions to PyInstaller 5.4

- Rok Mandeljc
- Brénainn Woodsend
- Efrem Braun
- Samuel T

2.17.24 Contributions to PyInstaller 5.3

- Rok Mandeljc
- Dan Yeaw
- Tim Gates

2.17.25 Contributions to PyInstaller 5.2

- Rok Mandeljc
- Brénainn Woodsend
- Florian Bruhin
- Zev Lee
- Highfire1
- Jasper Harrison
- KnockKnockWho
- Temerold
- relativisticelectron

2.17.26 Contributions to PyInstaller 5.1

- Rok Mandeljc
- Brénainn Woodsend
- Jasper Harrison
- byehack
- ARNTechnology
- James Gerity
- Kian-Meng Ang

2.17.27 Contributions to PyInstaller 5.0.1

- Rok Mandeljc
- Abdelhakim Qbaich
- Brénainn Woodsend
- Jasper Harrison

2.17.28 Contributions to PyInstaller 5.0

- Rok Mandeljc
- Brénainn Woodsend
- Jasper Harrison
- Starbuck5
- Chris Hillery
- Dan Yeaw
- eric15342335
-
- AdrianIssott
- Andreas Schwab
- Andrii Oriekhov
- Anssi Alahuhta
- Brian Teague
- Charlie Hayden
- Emil Berg
- Eric Missimer
- GoldinGuy
- James Gerity
- Melvin Wang
- Sapphire Becker
- dennisvang
- gentlegiantJGC
- johnthagen
- luc-x41
- wangling12

2.17.29 Contributions to PyInstaller 4.10

- Rok Mandeljc
- Brénainn Woodsend
- Andreas Schwab
- GoldinGuy
- Sapphire Becker
- dennisvang

2.17.30 Contributions to PyInstaller 4.9

- Rok Mandeljc - Core Developer
- Brénainn Woodsend - Core Developer
- Jasper Harrison - Core Develop, Maintainer, Release Manager
- gentlegiantJGC

2.17.31 Contributions to PyInstaller 4.8

- Rok Mandeljc - Core Developer
- Jasper Harrison - Core Develop, Maintainer, Release Manager
- Brénainn Woodsend - Core Developer
- Ankith, Safihre, luc-x41

2.17.32 Contributions to PyInstaller 4.7

- Rok Mandeljc - Core Developer
- Brénainn Woodsend - Core Developer
- Jasper Harrison - Core Develop, Maintainer, Release Manager

2.17.33 Contributions to PyInstaller 4.6

- Rok Mandeljc - Core Developer
- Brénainn Woodsend - Core Developer
- Jasper Harrison - Maintainer, Release Manager
- Anssi Alahutta, Dan Yeaw, Eric Missimer, Chris Hillery, Melvin Wang, wangling12, eric15342335

2.17.34 Contributions to PyInstaller 4.5.1

- Jasper Harrison - Maintainer, Release Manager
- ankith26

2.17.35 Contributions to PyInstaller 4.5

- Rok Mandeljc - Core Developer
- Brénainn Woodsend - Core Developer
- Jasper Harrison - Maintainer, Release Manager
- Dave Dykstra
- Andy Hobbs
- Nicholas Ollinger

2.17.36 Contributions to PyInstaller 4.4

- Rok Mandeljc - Core Developer
- Brénainn Woodsend - Core Developer
- Jasper Harrison - Core Developer, Maintainer, Release Manager
- Hartmut Goebel - Core Developer
- xoviat
- Chrisg2000
- Alex Gembe, James Duley, Jeffrey, Kenny Huynh, Maxim Mazurok, mozbugbox

2.17.37 Contributions to PyInstaller 4.3

- Rok Mandeljc - Core Developer
- Brénainn Woodsend - Core Developer
- Jasper Harrison (Legorooj) - Core Developer, Maintainer, Release Manager
- Hartmut Goebel, Core Developer, Maintainer
- xoviat
- Dan Yeaw, Bruno Oliveira, Maxim Kalinchenko, Max Mäusezahl, Olivier FAURAX, richardsheridan, memo-off

2.17.38 Contributions to PyInstaller 4.2

- Rok Mandeljc
- Hartmut Goebel - Core developer, maintainer and release manager.
- Legorooj - Core developer.
- Bryan A. Jones - Core developer and PyQt5-tamer.
- Mickaël Schoentgen
- Brénainn Woodsend
- Damien Elmes, Dan Yeaw, hdf, Diggy, Filip Gospodinov, Kyle Altendorf, Matt Simpson, Nathan Summers, Phoenix, Starbuck5, Tom Hu, rockwalrus

2.17.39 Contributions to PyInstaller 4.1

- Hartmut Goebel - Core developer, maintainer and release manager.
- Legorooj - Core developer.
- Bryan A. Jones - Core developer and PyQt5-tamer.
- Rok Mandeljc
- Mickaël Schoentgen
- Brénainn Woodsend

- Aaron Althauser, Alex, Andrew Nelson, Benedikt Brückmann, Brénainn Woodsend, Calin Culianu, Dan Yeaw, Ievgen Popovych, Loïc Messal, Łukasz Stolcman, Matt, Mohamed, Petrus, Riz, Riz Syed, Santi Santichaivekin, Sid Gupta, Victor Stinner, byehack, dcgloe, johnthagen, ozelikov,

2.17.40 Contributions to PyInstaller 4.0

- Hartmut Goebel - Core developer, maintainer and release manager.
- Legorooj - Core developer.
- Bryan A. Jones - Core developer and PyQt5-tamer.
- M Felt aka aixtools, jonnyhsu, Corey Dexter, Rok Mandeljc, Dan Yeaw, Florian Baumann, Ievgen Popovych, Ram Rachum, coreydexter, AndCycle, Dan Cutright, David Kiliani, David Maiden Mueller, FeralRobot, Frederico, Ilya Orson, ItsCinnabar, Juan Sotomayor, Matt M, Matteo Bertini, Michael Felt, Mohamed Feddad, Nehal J Wani, Or Groman, Sebastian Hohmann, Vaclav Dvorak, Ville Ilvonen, bwoodsend, eldadr, jeremyd2019, kraptor, seedgou.

2.17.41 Contributions to PyInstaller 3.6

- Hartmut Goebel - Core developer, maintainer and release manager.
- Bryan A. Jones - Core developer and PyQt5-tamer.
- Dan Yeaw, Amir Rossert, Hugo Martins, Felix Schwarz, Giuseppe Corbelli, HoLuLuLu, Jonathan Springer, Matt Khan, Min'an, Oracizan, Victor Stinner, Andres, Andrew Chow, Bernát Gábor, Charles Duffy, Chris, Chrisg2000, FranzPio, Lee Jeonghun, Lukasz Stolcman, Lyux, László Kiss Kollár, Mathias Lohne, Michael Felt, Noodle-Head, Ogi Moore, Patryk, RedFantom, Rémy Roy, Sean McGuire, Thomas Robitaille, Tim, Toby, Tuomo, V.Shkaberda, Vojtěch Drábek, Wilmar den Ouden, david, ethframe, lnv42, ripdog, satvidh, thisisivan-fong

2.17.42 Contributions to PyInstaller 3.5

- Hartmut Goebel - Core developer, maintainer and release manager.
- Bryan A. Jones - Core developer and PyQt5-tamer.
- Dave Cortesi, Kuisong Tong, melvyn2, Giuseppe Corbelli, Florian Bruhin, Amir Ramezani, Cesar Vandevelde, Paul Müller, Thomas Robitaille, zachbateman, Addison Elliott, Amir Rossert, AndCycle, Atomfighter10101, Chris Berthiaume, Craig Younkins (bot), Don Krueger, Edward Chen, Exane Server Team, Hannes, Iwan, Jakob Schnitzer, Janzert, Jendrik Seipp, Jonathan Springer, Kirill German, Laszlo Kiss-Kollar, Loran425, Lori J, M*C*O, Nikita Melentev, Peter Bittner, RedFantom, Roman, Roman Yurchak, Ruslan Kuprieiev, Spencer Brown, Suzumizaki, Tobias Gruetzmacher, Tobias V. Langhoff, TobiasRzepka, Tom Hacohen, Yuval Shkolar, cclauss, charlesoblack, djl197, matias morant, satejkhedekar, zhu

2.17.43 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
- Mickaël Schoentgen, Charles Nicholson, Jonathan Springer, Benoît Vinot, Brett Higgins, Dustin Spicuzza, Marco Nenciarini, Aaron Hampton, Cody Scot, Dave Cortesi, Helder Eijs, Innokenty Lebedev, Joshua Klein, Matthew Clapp, Misha Turnbull, ethframe, Amir Ramezani, Arthur Silva, Blue, Craig MacEachern, Cédric RICARD, Fredrik Ahlberg, Glenn Ramsey, Jack Mordaunt, Johann Bauer, Joseph Heck, Kyle Stewart, Lev Maximov, Luo Shawn, Marco Nenciarini, Mario Costa, Matt Reynolds, Matthieu Gautier, Michael Herrmann, Moritz Kassner, Natanael Arndt, Nejc Habjan, Paweł Kowalik, Pedro de Medeiros, Peter Conerly, Peter Würtz, Rémy Roy, Saurabh Yadav, Siva Prasad, Steve Peak, Steven M. Vascellaro, Steven M. Vascellaro, Suzumizaki-Kimitaka, ThomasV, Timothée Lecomte, Torsten Sommer, Weliton Freitas, Zhen Zhang, dimitriepirghie, lneuhaus, s3goat, satarsa,

2.17.44 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
- Dave Cortesi, David Hoese, John Daytona, Nejc Habjan, Addison Elliott, Bharath Upadhya, Bill Dengler, Chris Norman, Miles Erickson, Nick Dimou, Thomas Waldmann, David Weil, Placinta

2.17.45 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
-
- Adam Clark, AndCycle, Andreas Schiefer, Arthur Silva, Aswa Paul, Bharath Upadhya, Brian Teague, Charles Duffy, Chris Coutinho, Cody Scott, Czarek Tomczak, Dang Mai, Daniel Hyams, David Hoese, Eelco van Vliet, Eric Drechsel, Erik Bjäreholt, Hatem AlSum, Henry Senyondo, Jan Čapek, Jeremy T. Hetzel, Jonathan Dan, Julie Marchant, Luke Lee, Marc Abramowitz, Matt Wilkie, Matthew Einhorn, Michael Herrmann, Niklas Rosenstein, Philippe Ombredanne, Piotr Radkowski, Ronald Oussoren, Ruslan Kuprieiev, Segev Finer, Shengjing Zhu , Steve, Steven Noonan, Tibor Csonka, Till Bey, Tobias Gruetzmacher, (float)

2.17.46 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

2.17.47 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.

2.17.48 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

2.17.49 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.

2.17.50 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 - apply import hook.

2.17.51 Contributions to PyInstaller 2.1 and older

- Glenn Ramsey - PyQt5 import hook.
- David Cortesi - PyInstaller manual rewrite.
- Vaclav Smilauer - IPython import hook.
- Shane Hansen - Linux arm support.
- Bryan A. Jones - docutils, jinja2, sphinx, pytz, idlelib import hooks.
- Patrick Stewart <patstew at gmail dot com> - scipy import hook.
- Georg Schoelly <mail at georg-schoelly dot com> - storm ORM import hook.
- Vinay Sajip - zmq import hook.
- Martin Gamwell Dawids - AIX support.
- Hywel Richards - Solaris support.
- Brandyn White - packaged executable return code fix.
- Chien-An “Zero” Cho - PyUSB import hook.
- Daniel Hyams - h2py, wx.lib.pubsub import hooks.
- Hartmut Goebel - Python logging system for message output. Option –log-level.
- Florian 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 - sqlalchemy import hook.
- Seth Remington - PyGTK hook improvements.
- Marco Bonifazi - PyGTK hook improvements. PyOpenGL import hook.
- Jamie Kirkpatrick - paste import hook.
- Lorenzo Mancini - PyXML import hook fixes under Windows. OS X support. App bundle creation on OS X. Tkinter on OS X. Precompiled bootloaders for OS X.
- Lorenzo Berni - django import hook.
- Louai Al-Khanji - fixes with optparse module.
- Thomas Heller - set custom icon of Windows exe files.
- Eugene Prigorodov <eprigorodov at naumen dot ru> - KInterasDB import hook.
- David C. Morrill - vtkpython import hook.
- Alan James Salmoni - Tkinter interface to PyInstaller.

2.18 Man Pages

2.18.1 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, macOS, FreeBSD, OpenBSD, Solaris and AIX. Its main advantages over similar tools are that PyInstaller works with Python 3.8-3.11, 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

Positional Arguments

scriptname

Name of scriptfiles to be processed or exactly one .spec file. If a .spec file is specified, most options are unnecessary and are ignored.

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)

--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, DEPRECATION, ERROR, FATAL (default: INFO). Also settable via and overrides the PYI_LOG_LEVEL environment variable.

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 base-name)

--contents-directory CONTENTS_DIRECTORY For onedir builds only, specify the name of the directory in which all supporting files (i.e. everything except the executable itself) will be placed in. Use "." to re-enable old onedir layout without contents directory.

What To Bundle, Where To Search

–add-data SOURCE:DEST

Additional data files or directories containing data files to be added to the application. The argument value should be in form of "source:dest_dir", where source is the path to file (or directory) to be collected, dest_dir is the destination directory relative to the top-level application directory, and both paths are separated by a colon (:). To put a file in the top-level application directory, use . as a dest_dir. This option can be used multiple times.

–add-binary SOURCE:DEST

Additional binary files to be added to the executable. See the `--add-data` option for the format. 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. Equivalent to supplying the `pathex` argument in the spec file.

--hidden-import MODULENAME, --hiddenimport MODULENAME Name an import not visible in the code of the script(s). This option can be used multiple times.

--collect-submodules MODULENAME Collect all submodules from the specified package or module. This option can be used multiple times.

--collect-data MODULENAME, --collect-datas MODULENAME Collect all data from the specified package or module. This option can be used multiple times.

--collect-binaries MODULENAME Collect all binaries from the specified package or module. This option can be used multiple times.

--collect-all MODULENAME Collect all submodules, data files, and binaries from the specified package or module. This option can be used multiple times.

--copy-metadata PACKAGENAME Copy metadata for the specified package. This option can be used multiple times.

--recursive-copy-metadata PACKAGENAME Copy metadata for the specified package and all its dependencies. 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.

--splash IMAGE_FILE (EXPERIMENTAL) Add an splash screen with the image IMAGE_FILE to the application. The splash screen can display progress updates while unpacking.

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.

--optimize LEVEL Bytecode optimization level used for collected python modules and scripts. For details, see the section "Bytecode Optimization Level" in PyInstaller manual.

--python-option PYTHON_OPTION Specify a command-line option to pass to the Python interpreter at runtime. Currently supports "v" (equivalent to "–debug imports"), "u", "W <warning control>", "X <xoption>", and "hash_seed=<value>". For details, see the section "Specifying Python Interpreter Options" in PyInstaller manual.

-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 has 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 this also triggers building a Mac OS .app bundle. On Windows this option is automatically set if the first script is a '.pyw' file. This option is ignored on *NIX systems.

–hide-console {minimize-late,minimize-early,hide-early,hide-late}

Windows only: in console-enabled executable, have bootloader automatically hide or minimize the console window if the program owns the console window (i.e., was not launched from an existing console window).

-i <FILE.ico or FILE.exe,ID or FILE.icns or Image or "NONE">, --icon <FILE.ico or FILE.exe,ID or FILE.icns or Imag
FILE.ico: apply the 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. If an image file is entered that isn't in the platform format (ico on Windows, icns on Mac), PyInstaller tries to use Pillow to translate the icon into the correct format (if Pillow is installed). Use "NONE" to not apply any icon, thereby making the OS show some default (default: apply PyInstaller's icon). This option can be used multiple times.

--disable-windowed-traceback Disable traceback dump of unhandled exception in windowed (noconsole) mode (Windows and macOS only), and instead display a message that this feature is disabled.

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 that will request elevation upon application start.

--uac-uiaccess Using this option allows an elevated application to work with Remote Desktop.

Mac Os Specific Options

--argv-emulation Enable argv emulation for macOS app bundles. If enabled, the initial open document/URL event is processed by the bootloader and the passed file paths or URLs are appended to sys.argv.

--osx-bundle-identifier BUNDLE_IDENTIFIER Mac OS .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)

--target-architecture ARCH, --target-arch ARCH Target architecture (macOS only; valid values: x86_64, arm64, universal2). Enables switching between universal2 and single-arch version of frozen application (provided python installation supports the target architecture). If not target architecture is not specified, the current running architecture is targeted.

--codesign-identity IDENTITY Code signing identity (macOS only). Use the provided identity to sign collected binaries and generated executable. If signing identity is not provided, ad- hoc signing is performed instead.

--osx-entitlements-file FILENAME Entitlements file to use when code-signing the collected binaries (macOS only).

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. Note that on POSIX systems, PyInstaller's bootloader does NOT perform shell-style environment variable expansion on the given path string. Therefore, using environment variables (e.g., ~ or `$HOME`) in path will NOT work.

--bootloader-ignore-signals Tell the bootloader to ignore signals rather than forwarding them to the child process. Useful in situations where for example a supervisor process signals

both the bootloader and the 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

`pyi-makespec`(1), The PyInstaller Manual https://pyinstaller.readthedocs.io/, Project Homepage http://www.pyinstaller.org

2.18.2 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

Positional Arguments

scriptname

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, DEPRECATION, ERROR, FATAL (default: INFO). Also settable via and overrides the PYI_LOG_LEVEL environment variable.

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)

--contents-directory CONTENTS_DIRECTORY For onedir builds only, specify the name of the directory in which all supporting files (i.e. everything except the executable itself) will be placed in. Use "." to re-enable old onedir layout without contents directory.

What To Bundle, Where To Search

–add-data SOURCE:DEST

Additional data files or directories containing data files to be added to the application. The argument value should be in form of "source:dest_dir", where source is the path to file (or directory) to be collected, dest_dir is the destination directory relative to the top-level application directory, and both paths are separated by a colon (:). To put a file in the top-level application directory, use . as a dest_dir. This option can be used multiple times.

–add-binary SOURCE:DEST

Additional binary files to be added to the executable. See the `--add-data` option for the format. 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. Equivalent to supplying the `pathex` argument in the spec file.

--hidden-import MODULENAME, --hiddenimport MODULENAME Name an import not visible in the code of the script(s). This option can be used multiple times.

--collect-submodules MODULENAME Collect all submodules from the specified package or module. This option can be used multiple times.

--collect-data MODULENAME, --collect-datas MODULENAME Collect all data from the specified package or module. This option can be used multiple times.

--collect-binaries MODULENAME Collect all binaries from the specified package or module. This option can be used multiple times.

--collect-all MODULENAME Collect all submodules, data files, and binaries from the specified package or module. This option can be used multiple times.

--copy-metadata PACKAGENAME Copy metadata for the specified package. This option can be used multiple times.

--recursive-copy-metadata PACKAGENAME Copy metadata for the specified package and all its dependencies. 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.

--splash IMAGE_FILE (EXPERIMENTAL) Add an splash screen with the image IMAGE_FILE to the application. The splash screen can display progress updates while unpacking.

How To Generate

-d {all,imports,bootloader,noarchive}, –debug {all,imports,bootloader,noarchive}

R|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.

--optimize LEVEL Bytecode optimization level used for collected python modules and scripts. For details, see the section “Bytecode Optimization Level” in PyInstaller manual.

--python-option PYTHON_OPTION Specify a command-line option to pass to the Python interpreter at runtime. Currently supports “v” (equivalent to “–debug imports”), “u”, “W <warning control>”, “X <xoption>”, and “hash_seed=<value>”. For details, see the section “Specifying Python Interpreter Options” in PyInstaller manual.

-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 has 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 this also triggers building a Mac OS .app bundle. On Windows this option is automatically set if the first script is a ‘.pyw’ file. This option is ignored on *NIX systems.

–hide-console {minimize-late,minimize-early,hide-early,hide-late}

Windows only: in console-enabled executable, have bootloader automatically hide or minimize the console window if the program owns the console window (i.e., was not launched from an existing console window).

-i <FILE.ico or FILE.exe,ID or FILE.icns or Image or “NONE”>, --icon <FILE.ico or FILE.exe,ID or FILE.icns or Imag
FILE.ico: apply the 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. If an image file is entered that isn't in the platform format (ico on Windows, icns on Mac), PyInstaller tries to use Pillow to translate the icon into the correct format (if Pillow is installed). Use "NONE" to not apply any icon, thereby making the OS show some default (default: apply PyInstaller's icon). This option can be used multiple times.

--disable-windowed-traceback Disable traceback dump of unhandled exception in windowed (noconsole) mode (Windows and macOS only), and instead display a message that this feature is disabled.

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 that will request elevation upon application start.

--uac-uiaccess Using this option allows an elevated application to work with Remote Desktop.

Mac Os Specific Options

--argv-emulation Enable argv emulation for macOS app bundles. If enabled, the initial open document/URL event is processed by the bootloader and the passed file paths or URLs are appended to sys.argv.

--osx-bundle-identifier BUNDLE_IDENTIFIER Mac OS .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)

--target-architecture ARCH, --target-arch ARCH Target architecture (macOS only; valid values: x86_64, arm64, universal2). Enables switching between universal2 and single-arch version of frozen application (provided python installation supports the target architecture). If not target architecture is not specified, the current running architecture is targeted.

--codesign-identity IDENTITY Code signing identity (macOS only). Use the provided identity to sign collected binaries and generated executable. If signing identity is not provided, ad- hoc signing is performed instead.

--osx-entitlements-file FILENAME Entitlements file to use when code-signing the collected binaries (macOS only).

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. Note that on POSIX systems, PyInstaller's bootloader does NOT perform shell-style environment variable expansion on the given path string. Therefore, using environment variables (e.g., ~ or `$HOME`) in path will NOT work.

--bootloader-ignore-signals Tell the bootloader to ignore signals rather than forwarding them to the child process. Useful in situations where for example a supervisor process signals both the bootloader and the 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`(1), The PyInstaller Manual https://pyinstaller.readthedocs.io/, Project Homepage http://www.pyinstaller.org

2.19 Development Guide

2.19.1 Quickstart

- 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.

  Reformatting code without functional changes will generally not be accepted (for rationale 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.)

2.19.2 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*

2.19.3 Coding conventions

The PyInstaller project follows the **PEP 8** Style Guide for Python Code for new code. It uses yapf to do the bulk of the formatting (mostly putting spaces in the correct places) automatically and ruff to validate **PEP 8** rules which yapf doesn't cover.

Before submitting changes to PyInstaller, please check your code with both tools.

To install them run:

```
pip install ruff toml yapf==0.32.0
```

Reformat your code automatically with yapf:

```
yapf -rip .
```

Then manually adjust your code based on any suggestions given by ruff:

```
ruff --fix .
```

Please abstain from reformatting existing code, even it 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.

2.19.4 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, ...:

```
pip download --no-deps https://github.com/pyinstaller/pyinstaller/archive/develop.
↪zip
unzip develop.zip
cd pyinstaller-develop/
```

   ... or using git:

```
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:

```
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.:

```
pytest tests/unit -k test_collect_submod_all_included
```

4. Run the test-suite:

```
pytest 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:

```
pip install -U -r tests/requirements-libraries.txt
pytest 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 macOS) and `appveyor.yml` (for Windows).

2.19.5 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).
- Reformatting 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.
- Use `git rebase -i` to sort, squash, and fixup commits prior to submitting the pull-request. Make it a readable history, easy to understand what you've done.

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 @5c1628e or @73d7710.

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.
```

- Do not start your commit message with a hash-mark (#) as git some git commands may dismiss these message. (See this discussion. for details.)

[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.

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:

```
# 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/
- 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/
- http://365git.tumblr.com/post/3308646748/writing-git-commit-messages
- http://www.catb.org/esr/dvcs-migration-guide.html
- https://git.dthompson.us/presentations.git/tree/HEAD:/happy-patching
- and other places.

2.19.6 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, 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 -r doc/requirements.txt
cd doc
make html
xdg-open _build/html/index.html
```

Please watch out for any warnings and errors while building the documentation. In your browser check if the markup is valid prior to pushing your changes and creating the pull-request. Please also run:

```
make clean
...
make 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*[0] in addition 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 @a1b2c3d.

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

[0] Defined in `doc/_extensions/pyi_sphinx_roles.py`

reStructuredText Cheat-sheet

- Combining markup and links:

```
The easiest way to install PyInstaller is using |pip|_::

.. |pip| replace:: :command:`pip`
.. _pip: https://pip.pypa.io/
```

2.19.7 Creating Pull-Requests

Example

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

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

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

    ```
    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):

     ```
     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:

```
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:

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

2.19.8 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.

  Our categories are: `feature`, `bugfix`, `breaking` (breaking changes), `deprecation`, `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:

[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.

  - 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 of 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.

2.19.9 pyenv and PyInstaller

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

- clone pyenv repository:

```
git clone https://github.com/yyuu/pyenv.git ~/.pyenv
```

- clone virtualenv plugin:

```
git clone https://github.com/yyuu/pyenv-virtualenv.git \
          ~/.pyenv/plugins/pyenv-virtualenv
```

- 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 PyInstaller 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`

2.19.10 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:[1]

***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.

2.20 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.

PYTHON MODULE INDEX

p

pyi_splash, 81
PyInstaller.compat, 91
PyInstaller.isolated, 100
PyInstaller.utils.hooks, 92
PyInstaller.utils.hooks.conda, 98

INDEX

Symbols

__init__() (*Splash method*), 44
-D
    command line option, 13
-F
    command line option, 13
--add-binary SOURCE:DEST
    command line option, 14
--add-data SOURCE:DEST
    command line option, 14
--additional-hooks-dir HOOKSPATH
    command line option, 14
--argv-emulation
    command line option, 16
--bootloader-ignore-signals
    command line option, 16
--clean
    command line option, 13
--codesign-identity IDENTITY
    command line option, 16
--collect-all MODULENAME
    command line option, 14
--collect-binaries MODULENAME
    command line option, 14
--collect-data MODULENAME
    command line option, 14
--collect-datas MODULENAME
    command line option, 14
--collect-submodules MODULENAME
    command line option, 14
--console
    command line option, 15
--contents-directory CONTENTS_DIRECTORY
    command line option, 13
--copy-metadata PACKAGENAME
    command line option, 14
--debug {all,imports,bootloader,noarchive}
    command line option, 15
--disable-windowed-traceback
    command line option, 15
--distpath DIR
    command line option, 13
--exclude-module EXCLUDES
    command line option, 14
--help
    command line option, 13
--hidden-import MODULENAME
    command line option, 14
--hiddenimport MODULENAME
    command line option, 14
--hide-console {minimize-late,minimize-early,hide-early,hi
    command line option, 15
--icon <FILE.ico or FILE.exe,ID or
        FILE.icns or Image or "NONE">
    command line option, 15
--log-level LEVEL
    command line option, 13
--manifest <FILE or XML>
    command line option, 16
--name NAME
    command line option, 13
--noconfirm
    command line option, 13
--noconsole
    command line option, 15
--noupx
    command line option, 15
--nowindowed
    command line option, 15
--onedir
    command line option, 13
--onefile
    command line option, 13
--optimize LEVEL
    command line option, 15
--osx-bundle-identifier BUNDLE_IDENTIFIER
    command line option, 16
--osx-entitlements-file FILENAME
    command line option, 16
--paths DIR
    command line option, 14
--python-option PYTHON_OPTION
    command line option, 15
--recursive-copy-metadata PACKAGENAME

command line option, 14
--resource RESOURCE
command line option, 16
--runtime-hook RUNTIME_HOOKS
command line option, 14
--runtime-tmpdir PATH
command line option, 16
--specpath DIR
command line option, 13
--splash IMAGE_FILE
command line option, 14
--strip
command line option, 15
--target-arch ARCH
command line option, 16
--target-architecture ARCH
command line option, 16
--uac-admin
command line option, 16
--uac-uiaccess
command line option, 16
--upx-dir UPX_DIR
command line option, 13
--upx-exclude FILE
command line option, 15
--version
command line option, 13
--version-file FILE
command line option, 16
--windowed
command line option, 15
--workpath WORKPATH
command line option, 13
-c
command line option, 15
-d {all,imports,bootloader,noarchive}
command line option, 15
-h
command line option, 13
-i <FILE.ico or FILE.exe,ID or FILE.icns or Image or "NONE">
command line option, 15
-m <FILE or XML>
command line option, 16
-n NAME
command line option, 13
-p DIR
command line option, 14
-r RESOURCE
command line option, 16
-s
command line option, 15
-v
command line option, 13
-w
command line option, 15
-y
command line option, 13

B

base_prefix (*in module PyInstaller.compat*), 91

C

call() (*in module PyInstaller.isolated*), 100
call() (*Python method*), 102
CC, 117
check_requirement() (*in module PyInstaller.utils.hooks*), 92
close() (*in module pyi_splash*), 81
collect_all() (*in module PyInstaller.utils.hooks*), 93
collect_data_files() (*in module PyInstaller.utils.hooks*), 94
collect_delvewheel_libs_directory() (*in module PyInstaller.utils.hooks*), 97
collect_dynamic_libs() (*in module PyInstaller.utils.hooks*), 95
collect_dynamic_libs() (*in module PyInstaller.utils.hooks.conda*), 100
collect_entry_point() (*in module PyInstaller.utils.hooks*), 96
collect_submodules() (*in module PyInstaller.utils.hooks*), 94
command line option
-D, 13
-F, 13
--add-binary SOURCE:DEST, 14
--add-data SOURCE:DEST, 14
--additional-hooks-dir HOOKSPATH, 14
--argv-emulation, 16
--bootloader-ignore-signals, 16
--clean, 13
--codesign-identity IDENTITY, 16
--collect-all MODULENAME, 14
--collect-binaries MODULENAME, 14
--collect-data MODULENAME, 14
--collect-datas MODULENAME, 14
--collect-submodules MODULENAME, 14
--console, 15
--contents-directory CONTENTS_DIRECTORY, 13
--copy-metadata PACKAGENAME, 14
--debug {all,imports,bootloader,noarchive}, 15
--disable-windowed-traceback, 15
--distpath DIR, 13
--exclude-module EXCLUDES, 14
--help, 13
--hidden-import MODULENAME, 14

--hiddenimport MODULENAME, 14
--hide-console {minimize-late,minimize-early,hide-early,hide-late}, 15
--icon <FILE.ico or FILE.exe,ID or FILE.icns or Image or "NONE">, 15
--log-level LEVEL, 13
--manifest <FILE or XML>, 16
--name NAME, 13
--noconfirm, 13
--noconsole, 15
--noupx, 15
--nowindowed, 15
--onedir, 13
--onefile, 13
--optimize LEVEL, 15
--osx-bundle-identifier BUNDLE_IDENTIFIER, 16
--osx-entitlements-file FILENAME, 16
--paths DIR, 14
--python-option PYTHON_OPTION, 15
--recursive-copy-metadata PACKAGENAME, 14
--resource RESOURCE, 16
--runtime-hook RUNTIME_HOOKS, 14
--runtime-tmpdir PATH, 16
--specpath DIR, 13
--splash IMAGE_FILE, 14
--strip, 15
--target-arch ARCH, 16
--target-architecture ARCH, 16
--uac-admin, 16
--uac-uiaccess, 16
--upx-dir UPX_DIR, 13
--upx-exclude FILE, 15
--version, 13
--version-file FILE, 16
--windowed, 15
--workpath WORKPATH, 13
-c, 15
-d {all,imports,bootloader,noarchive}, 15
-h, 13
-i <FILE.ico or FILE.exe,ID or FILE.icns or Image or "NONE">, 15
-m <FILE or XML>, 16
-n NAME, 13
-p DIR, 14
-r RESOURCE, 16
-s, 15
-v, 13
-w, 15
-y, 13
scriptname, 13
commit (*role*), 226
copy_metadata() (*in module PyInstaller.utils.hooks*), 96

D

decorate() (*in module PyInstaller.isolated*), 101
Distribution (*class in PyInstaller.utils.hooks.conda*), 99
distribution() (*in module PyInstaller.utils.hooks.conda*), 98

E

environment variable
CC, 117
OBJECT_MODE, 26, 117
PYTHONDEVMODE, 132
PYTHONHASHSEED, 86
PYTHONMALLOC, 132
PYTHONOPTIMIZE, 51
PYTHONPATH, 37, 128
PYTHONUTF8, 129
PYTHONWARNDEFAULTENCODING, 123
eval_statement() (*in module PyInstaller.utils.hooks*), 92
exec_statement() (*in module PyInstaller.utils.hooks*), 92
EXTENSION_SUFFIXES (*in module PyInstaller.compat*), 91

F

files() (*in module PyInstaller.utils.hooks.conda*), 99

G

get_homebrew_path() (*in module PyInstaller.utils.hooks*), 97
get_hook_config() (*in module PyInstaller.utils.hooks*), 103
get_module_attribute() (*in module PyInstaller.utils.hooks*), 95
get_module_file_attribute() (*in module PyInstaller.utils.hooks*), 95
get_package_paths() (*in module PyInstaller.utils.hooks*), 96

I

include_or_exclude_file() (*in module PyInstaller.utils.hooks*), 97
is_aix (*in module PyInstaller.compat*), 91
is_alive() (*in module pyi_splash*), 81
is_cygwin (*in module PyInstaller.compat*), 91
is_darwin (*in module PyInstaller.compat*), 91
is_freebsd (*in module PyInstaller.compat*), 91
is_linux (*in module PyInstaller.compat*), 91
is_module_or_submodule() (*in module PyInstaller.utils.hooks*), 94
is_module_satisfies() (*in module PyInstaller.utils.hooks*), 93

is_openbsd (*in module PyInstaller.compat*), 91
is_package() (*in module PyInstaller.utils.hooks*), 94
is_solar (*in module PyInstaller.compat*), 91
is_venv (*in module PyInstaller.compat*), 91
is_win (*in module PyInstaller.compat*), 91
issue (*role*), 226

L

locate() (*PackagePath method*), 99

M

module
    pyi_splash, 81
    PyInstaller.compat, 91
    PyInstaller.isolated, 100
    PyInstaller.utils.hooks, 92
    PyInstaller.utils.hooks.conda, 98

O

OBJECT_MODE, 26, 117

P

package_distribution() (*in module PyInstaller.utils.hooks.conda*), 98
PackagePath (*class in PyInstaller.utils.hooks.conda*), 99
pyi_splash
    module, 81
PyInstaller.compat
    module, 91
PyInstaller.isolated
    module, 100
PyInstaller.utils.hooks
    module, 92
PyInstaller.utils.hooks.conda
    module, 98
Python (*class in PyInstaller.isolated*), 102
Python Enhancement Proposals
    PEP 0508, 92, 93
    PEP 239, 151
    PEP 263, 133
    PEP 302, 79, 125
    PEP 405, 21
    PEP 451, 125
    PEP 527, 179
    PEP 552, 169
    PEP 8, 221, 222
PYTHONDEVMODE, 132
PYTHONHASHSEED, 86
PYTHONMALLOC, 132
PYTHONOPTIMIZE, 51
PYTHONPATH, 37, 128
PYTHONUTF8, 129
PYTHONWARNDEFAULTENCODING, 123

R

requires() (*in module PyInstaller.utils.hooks.conda*), 99

S

scriptname
    command line option, 13

U

update_text() (*in module pyi_splash*), 81

W

walk_dependency_tree() (*in module PyInstaller.utils.hooks.conda*), 99