PyInstaller Manual

Version:	PyInstaller 3.0.dev8+f1a8933.mod
Homepage:	http://www.pyinstaller.org
Contact:	pyinstaller@googlegroups.com
Authors:	David Cortesi
	based on structure by Giovanni Bajo & William Caban
	based on Gordon McMillan's manual
Copyright:	This document has been placed in the public domain.

Contents

In Brief

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

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

What's New This Release

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

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

Requirements

Windows

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

PyInstaller requires the PyWin32 or pypiwin32 Python extension for Windows. If you install *PyInstaller* using pip, and PyWin32 is not found, pypiwin32 is automatically installed.

The pip-Win package is also recommended but not required.

Mac OS X

PyInstaller runs in Mac OS X 10.6 (Snow Leopard) or newer. It builds 64-bit executables by default, but can create 32-bit executables. It can build graphical windowed apps (apps that do not use a terminal window).

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. This is typically found in the distribution-package `binutils`.

AIX, Solaris, and FreeBSD

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

Before using *PyInstaller* in these systems you must compile a bootloader; see Building the Bootloader.

License

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

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

For updated information or clarification see our FAQ at the PyInstaller home page.

How To Contribute

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

How to Install *PyInstaller*

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

```
pip install pyinstaller
```

or upgrade to a newer version:

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

Installing in Windows

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

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

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

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

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

```
pip install PyInstaller
```

Once it is installed, to use *PyInstaller*,

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

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

Installing in Mac OS X

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

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

Installing from the archive

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

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

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

Verifying the installation

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

```
pyinstaller --version
```

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

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

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

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

Installed commands

The complete installation places these commands on the execution path:

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

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

What *PyInstaller* Does and How It Does It

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

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

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

    pyinstaller myscript.py

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

    pyinstaller --onefile --windowed myscript.py

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

> ***Note***
>
> The output of *PyInstaller* is specific to the active operating system and the active version of Python. This means that to prepare a distribution for:
>
> - a different OS
> - a different version of Python
> - a 32-bit or 64-bit OS
>
> you run *PyInstaller* on that OS, under that version of Python. The Python interpreter that executes *PyInstaller* is part of the bundle, and it is specific to the OS and the word size.

Analysis: Finding the Files Your Program Needs

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

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

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

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

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

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

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

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

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

Bundling to One Folder

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

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

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

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

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

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, Linux, Mac OS X, etc.). When the user launches your program, it is the bootloader that runs. The bootloader creates a temporary Python environment such that the Python interpreter will find all imported modules and libraries in the `myscript` folder.

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

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

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.

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 `_MEI`*xxxxxx*, 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.

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

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

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

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

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

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

Using a Console Window

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

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

Hiding the Source Code

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

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

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

Using PyInstaller

The syntax of the `pyinstaller` command is:

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

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

```
pyinstaller myscript.py
```

PyInstaller analyzes `myscript.py` and:

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

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

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

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

`pyinstaller myscript.spec`

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

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

or, on Windows,

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

Options

General Options

`-h, --help`	show this help message and exit
`-v, --version`	Show program version info and exit.
`--distpath=DIR`	Where to put the bundled app (default: ./dist)
`--workpath=WORKPATH`	Where to put all the temporary work files, .log, .pyz and etc. (default: ./build)
`-y, --noconfirm`	Replace output directory (default: SPECPATH/dist/SPECNAME) without asking for confirmation
`--upx-dir=UPX_DIR`	Path to UPX utility (default: search the execution path)

`-a, --ascii`	Do not include unicode encoding support (default: included if available)
`--clean`	Clean PyInstaller cache and remove temporary files before building.
`--log-level=LOGLEVEL`	Amount of detail in build-time console messages (default: INFO, choose one of DEBUG, INFO, WARN, ERROR, CRITICAL)

What to generate

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

What to bundle, where to search

`-p DIR, --paths=DIR`	A path to search for imports (like using PYTHONPATH). Multiple paths are allowed, separated by ':', or use this option multiple times
`--hidden-import=MODULENAME`	Name an import not visible in the code of the script(s). This option can be used multiple times.
`--additional-hooks-dir=HOOKSPATH`	An additional path to search for hooks. This option can be used multiple times.
`--runtime-hook=RUNTIME_HOOKS`	Path to a custom runtime hook file. A runtime hook is code that is bundled with the executable and is executed before any other code or module to set up special features of the runtime environment. This option can be used multiple times.
`--exclude-module=EXCLUDES`	Optional module or package (his Python names,not path names) that will be ignored (as thoughit was not found).This option can be used multiple times.
`--key=KEY`	The key used to encrypt Python bytecode.

How to generate

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

Windows and Mac OS X specific options

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

Windows specific options

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

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

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

Mac OS X specific options

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

Shortening the Command

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

```
pyinstaller --noconfirm --log-level=WARN \
    --onefile --nowindow \
    --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 ^
    --hidden-import=secret1 ^
    --hidden-import=secret2 ^
    --icon-file=..\MLNMFLCN.ICO ^
    myscript.spec
```

Using UPX

UPX is a free utility available for most operating systems. UPX compresses executable files and libraries, making them smaller, sometimes much smaller. UPX is available for most operating systems and can compress a large number of executable file formats. See the UPX home page for downloads, and for the list of supported executable formats. Development of UPX appears to have ended in September 2013, at which time it supported most executable formats except for 64-bit binaries for Mac OS X. UPX has no effect on those.

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

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

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

Encrypting Python Bytecode

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

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

Supporting Multiple Platforms

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

Supporting Multiple Python Environments

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

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

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

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

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

Supporting Multiple Operating Systems

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

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

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

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

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

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

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

Making Linux Apps Forward-Compatible

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

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

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

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

Capturing Windows Version Data

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

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

`pyi-grab_version` *executable_with_version_resource*

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

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

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

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

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

The elements of each tuple represent 16-bit values from most-significant to least-significant. For example the value `(2, 0, 4, 0)` resolves to `0002000000040000` in hex.

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

> `set_version` *version_text_file executable_file*

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

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

Building Mac OS X App Bundles

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

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

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

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

PyInstaller builds minimal versions of these elements.

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

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

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

Getting the Opened Document Names

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

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

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

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

Run-time Operation

Your app should run in a bundle exactly as it does when run from source. However, you might want to learn at run-time whether the app is running from source or "frozen" (bundled).

For example, you might have data files that, when running live, are found based on a module's `__file__` attribute. That will not work when the code is bundled.

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

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

Data files and folders of files can be included in the bundle. by editing the spec file; see Adding Files to the Bundle. The added files will be in the bundle folder.

The bootloader stores the absolute path to the bundle folder in `sys._MEIPASS`. For a one-folder bundle, this is the path to that folder, wherever the user may have put it. For a one-file bundle, this is the path to the `_MEIxxxxxx` temporary folder created by the bootloader (see How the One-File Program Works).

When your application needs access to a data file that is bundled with it, you get the path to the file with the following code:

```
import sys
import os
...
if getattr(sys, 'frozen', False):
    # we are running in a bundle
    basedir = sys._MEIPASS
else:
    # we are running in a normal Python environment
    basedir = os.path.dirname(os.path.abspath(__file__))
```

This code sets `basedir` to the path to the folder containing your script and any other files or folders bundled with it. When your program was not started by the bootloader, the standard Python variable `__file__` is the full path to the script now executing, and `os.path.dirname()` extracts the path to the folder that contains it. When bundled, `sys._MEIPASS` provides the path to bundle folder.

Using Spec Files

When you execute

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

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

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

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

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

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

These uses are covered in topics below.

You create a spec file using this command:

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

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

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

`pyinstaller` *options name*`.spec`

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

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

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

Spec File Operation

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

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

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

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

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

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

Adding Files to the Bundle

To add files to the bundle, you create a list that describes the files and supply it to the `Analysis` call.

Adding Data Files

You 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 names of the files in the bundled app at run-time.

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

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

```
     hiddenimports=...
     )
```

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`. This file will be copied into the bundle with name `README`.

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

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

The strings may use either `/` or `\` as the path separator character. You can specify input files using "glob" abbreviations. When the input is multiple files, the output string may be the name of a folder. For example to include all the `.mp3` files from a certain folder:

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

All files matching `/mygame/sfx/*.mp3` will be copied into the bundle and stored in a folder named `sfx`.

The path to the input file or folder may be absolute as in the first tuple, or relative as in the second. When it is relative, it is taken as relative to the location of the spec file.

You can also include the entire contents of a folder:

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

All files in `/mygame/data` will be copied recursively into a folder named `data` in the bundle.

Adding Binary Files

To add binary files, make a list of tuples that describe the files needed. Assign the list of tuples to the `binaries=` argument of Analysis.

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

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

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

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

Advanced Methods of Adding Files

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

Giving Run-time Python Options

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

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

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

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

For example modify the spec file this way:

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

Spec File Options for a Mac OS X Bundle

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

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

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

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

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

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

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

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

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

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

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

Multipackage Bundles

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

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

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

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

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

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

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

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

```
MERGE(*args)
```

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

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

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

Example MERGE spec file

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

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

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

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

```
foo_a = Analysis(['foo.py'],
        pathex=['/the/path/to/foo'],
        hiddenimports=[],
```

```
        hookspath=None)

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

Recipes and Examples for Specific Problems

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

- A more sophisticated way of collecting data files than the one shown above (Adding Files to the Bundle).

- Bundling a typical Django app.
- A use of a run-time hook to set the PyQt4 API level.
- A workaround for a multiprocessing constraint under Windows.

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

Finding out What Went Wrong

Build-time Messages

When the `Analysis` step runs, it produces error and warning messages. These display after the command line if the `--log-level` option allows it. Analysis also puts messages in a warnings file named `build/`*name*`/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 Python Errors

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

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

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

Getting Debug Messages

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

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

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

Getting Python's Verbose Imports

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

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

Helping PyInstaller Find Modules

Extending the Path

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

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

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

Listing Hidden Imports

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

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

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

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

Extending a Package's `__path__`

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

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

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

Once in a while, that's not enough.

Changing Runtime Behavior

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

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

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

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

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

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

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

Getting the Latest Version

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

If you have Git installed on your development system, you can use it together with pip to install the latest version of *PyInstaller* directly:

```
pip install -e git://github.com/pyinstaller/pyinstaller.git#egg=PyInstaller
```

Asking for Help

When none of the above suggestions help, do ask for assistance on the PyInstaller Email List.

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

Advanced Topics

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

The Bootstrap Process in Detail

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

Bootloader

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

What happens during execution of bootloader:

A. First process: bootloader starts.

    1. If one-file mode, extract bundled files to *temppath*`_MEI`*xxxxxx*
    2. Set/unset various environment variables, e.g. override LD_LIBRARY_PATH on Linux or LIBPATH on AIX; unset DYLD_LIBRARY_PATH on OSX.
    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`.

The TOC and Tree Classes

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

TOC Class (Table of Contents)

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

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

(*name*, *path*, *typecode*)

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

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

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

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

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

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

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

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

The Tree Class

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

`Tree(`*root*`, prefix=`*run-time-folder*`, excludes=`*match*`)`

- The *root* argument is a path string to a directory. It may be absolute or relative to the spec file directory.
- The *prefix* argument, if given, is a name for a subfolder within the run-time folder to contain the tree files. If you omit *prefix* or give `None`, the tree files will be at the top level of the run-time folder.
- The *excludes* argument, if given, is a list of one or more strings that match files in the *root* that should be omitted from the Tree. An item in the list can be either:
    - a name, which causes files or folders with this basename to be excluded
    - `*.ext`, which causes files with this extension to be excluded

For example:

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

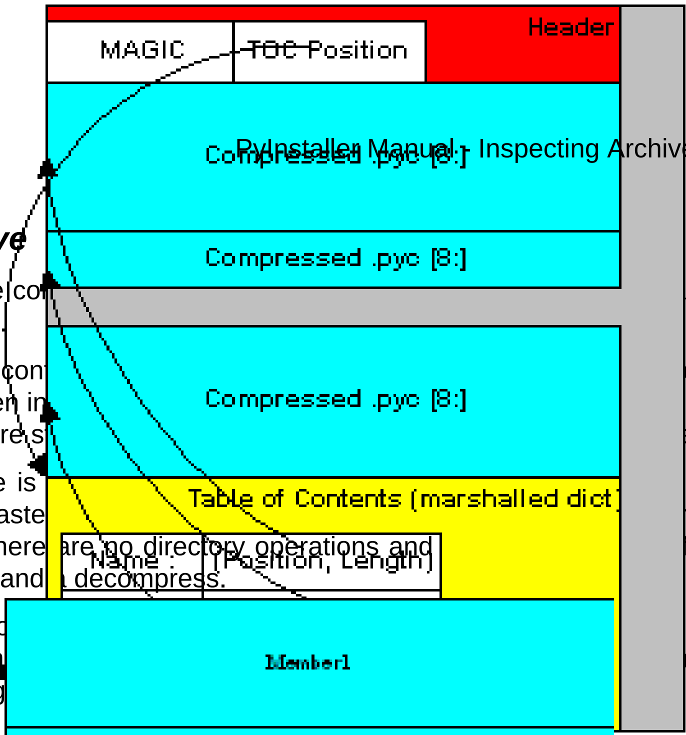
This creates `extras_toc` as a TOC object that lists all files from the relative path `../src/extras`, omitting those that have the basename (or are in a folder named) `tmp` or that have the type `.pyc`.

Each tuple in this TOC has:

- A *typecode* of `DATA`,
- A *path* consisting of a complete, absolute path to one file in the *root* folder,
- A *name* consisting of the filename of this file, or, if you specify a *prefix*, the *name* is *prefix/filename*.

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.



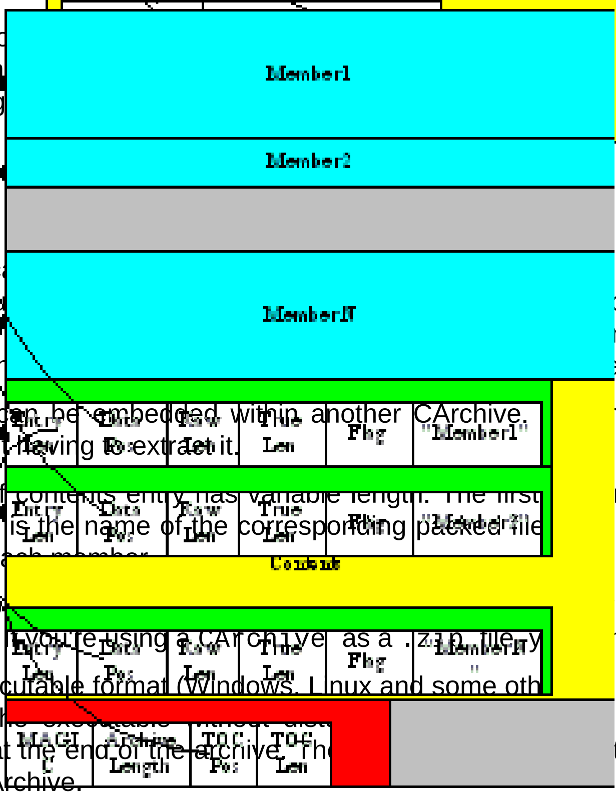
ZlibArchive

A ZlibArchive co ss invocation in a spec file creates a ZlibArchive.

The table of con ciates a key, which is a member's name as given in gth in the ZlibArchive. All parts of a ZlibArchive are s ependent.

A ZlibArchive is Even with maximum compression this works faste s.path, there's a lookup in the dictionary. There are no directory operations and file is already open). There's just a seek, a read and a decompress.

A Python erro entry was created (the `__file__` attribute from the archive). This will not tell your user anything make sense of it.



CArchive

A CArchive c p file. They are easy to create in Python and easy to u o another file, such as an ELF and COFF executable. ntents at the end of the file, followed only by a cookie th e the archive itself starts.

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

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

There is also pe codes are used by the self-extracting executables. If you're using a CArchive as a .zip file, y n't need to worry about the code.

The ELF executable format (Windows, Linux and some oth lows arbitrary data to be concatenated to the end of the executable without dist For this reason, a CArchive's Table of Contents is at the end of the archive. The tself 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 `outPYZ.pyz` archive inside it.

U

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

X *name*

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

Q

Quit.

The `pyi-archive_viewer` command has these options:

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

Inspecting Executables

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

`pyi-bindepend` *executable_or_dynamic_library*

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

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

Understanding PyInstaller Hooks

> ***Note***
>
> THE FOLLOWING IS THE TEXT FROM THE 2.1 MANUAL.
>
> IT NEEDS TO BE REWRITTEN FOR THE NEW HOOKS API OF VERSION 3.0.
>
> DO NOT ATTEMPT TO BUILD OR EDIT A HOOK BASED ON THIS TEXT.

In summary, a "hook" file tells *PyInstaller* about hidden imports called by a particular module. The name of the hook file is `hook-<module>.py` where "<module>" is the name of a script or imported module that will be found by Analysis. You should browse through the existing hooks in the `hooks` folder of the *PyInstaller* distribution folder, if only to see the names of the many supported imports.

For example `hook-cPickle.py` is a hook file telling about hidden imports used by the module `cPickle`. When your script has `import cPickle` the Analysis will note it and check for a hook file `hook-cPickle.py`.

Typically a hook module has only one line; in `hook-cPickle.py` it is

```
hiddenimports = ['copy_reg', 'types', 'string']
```

assigning a list of one or more module names to `hiddenimports`. These module names are added to the Analysis list exactly as if the script being analyzed had imported them by name.

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

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

If you successfully hook a publicly distributed module in this way, please send us the hook file so we can make it available to others.

Hooks in Detail

A hook is a module named `hook-` *fully.qualified.import.name* `.py` in the `hooks` folder of the *PyInstaller* folder (or in a folder specified with `--additional-hooks-dir`).

A hook is executable Python code that should define one or more of the following several global names:

> ***Note***
>
> A hook is just a normal Python script. So you can do all things like testing `sys.version` and adjust e.g. `hiddenimports` based on that.

`excludedimports`

A list of module names (relative or absolute) that the hooked module excludes in some opaque way. These names reduce the list of imported modules created by scanning the code. Example:

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

`hiddenimports`

A list of module names (relative or absolute) that the hooked module imports in some opaque way. These names extend the list of imported modules created by scanning the code. Example:

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

A way to simplify adding all submodules of a package is to use:

```
from PyInstaller.utils.hooks import collect_submodules
hiddenimports = collect_submodules('package')
```

For an example see `hook-docutils.py` in the hooks folder.

Note: We suggest always using the fully qualified name `PyInstaller.utils.hooks` for importing the hook utilities. This avoids some pitfalls when implementing hooks for sub-modules.

`datas`

A list of globs of files or directories to bundle as datafiles. For each glob, a destination directory is specified.

Example:

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

This will copy all files matching *education_*.png* into the subdirectory *icons*, and recursively (because of the `*` wildcard) copy the content of */usr/share/libsmi/mibs* into *mibs*.

A way to simplify collecting a folder of files is to use:

```
from PyInstaller.utils.hooks import collect_data_files
datas = collect_data_files('package_name')
```

to collect all package-related data files into a folder *package_name* in the app bundle. For an example see hook-pytz.py in the hooks folder.

binaries

A list of globs of files or directories to bundle as binaries. Binaries is a special case of `datas` in that PyInstaller will check if they depend on other possible dynamic libraries. Otherwise it looks the same.

Example:

```
binaries = [
     ('/usr/lib/lib*.so', 'libs'),
     ('C:\\Windows\\System32\\*.dll', 'dlls'),
     ]
```

attrs

A list of (*name* , *value*) pairs (where value is normally meaningless).

This will set the module-attribute *name* to *value* for each pair in the list. The value is usually unimportant because the modules are not executed.

The main purpose is so that ImportTracker will not issue spurious warnings when the rightmost node in a dotted name turns out to be an attribute in a package, instead of a missing submodule. For an example see the hook file `hook-xml.sax.py`.

`def hook(mod):`

> ***Note***
>
> The need to use this should be rare. Instead, try to use the global names described above first. This will keep the hook's code simple.

Defines a function that takes a `Module` object. It must return a `Module` object, possibly the same one unchanged, or a modified one. A `Module` object is an instance of the class `PyInstaller.depend.modules.Module()` which you can read. If defined, `hook(mod)` is called before *PyInstaller* processed the global names described above.

This function is supported to handle cases like dynamic modification of a package's `__path__` variable. A static list of names won't suffice because the new entry on `__path__` may well require computation. See `hook-win32com.py` in the hooks folder for an example.

Building the Bootloader

PyInstaller comes with binary bootloaders for most platforms in the `bootloader` folder of the distribution folder. For most cases, these precompiled bootloaders are all you need.

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

For

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

This will produce the bootloader executables,

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

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

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

Development tools

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

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

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

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

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

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

Building for Windows

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

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

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

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

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

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

Change to the `bootloader` subdirectory. Run:

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

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

Building for LINUX

By default, the bootloaders on Linux are LSB binaries.

LSB is a set of open standards that should increase compatibility among Linux distributions. *PyInstaller* produces a bootloader as an LSB binary in order to increase compatibility for packaged applications among distributions.

Note: LSB version 4.0 is required for successfull building of bootloader.

On Debian- and Ubuntu-based distros, you can install LSB 4.0 tools by adding the following repository to the sources.list file:

```
deb http://ftp.linux-foundation.org/pub/lsb/repositories/debian lsb-4.0 main
```

then after having update the apt repository:

```
sudo apt-get update
```

you can install LSB 4.0:

```
sudo apt-get install lsb lsb-build-cc
```

Most other distributions contain only LSB 3.0 in their software repositories and thus LSB build tools 4.0 must be downloaded by hand. From Linux Foundation download LSB sdk 4.0 for your architecture.

Unpack it by:

```
tar -xvzf lsb-sdk-4.0.3-1.ia32.tar.gz
```

To install it run:

```
cd lsb-sdk
./install.sh
```

After having installed the LSB tools, you can follow the standard building instructions.

NOTE: if for some reason you want to avoid LSB compilation, you can do so by specifying --no-lsb on the waf command line, as follows:

```
python waf configure --no-lsb build install
```

This will also produce `support/loader/YOUR_OS/run`, `support/loader/YOUR_OS/run_d`, `support/loader/YOUR_OS/runw` and `support/loader/YOUR_OS/runw_d`, but they will not be LSB binaries.