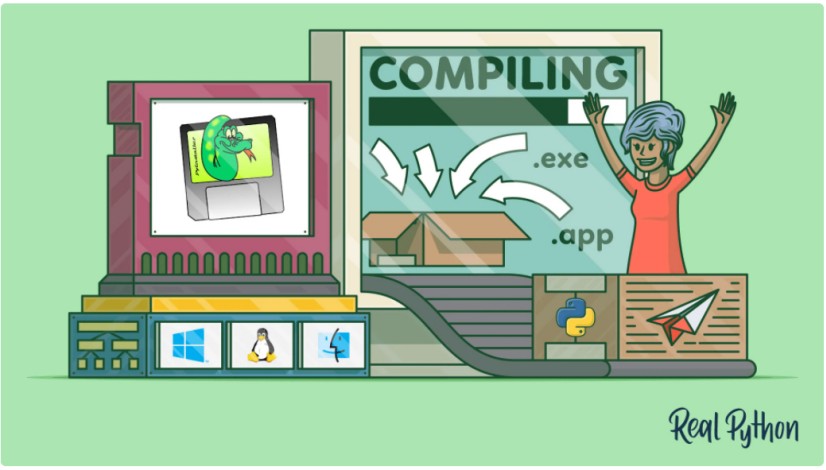


Using PyInstaller to Easily Distribute Python Applications

by Luke Lee Mar 06, 2019 23 Comments best-practices intermediate

Mark as Completed

Tweet Share Email

Table of Contents

Are you jealous of Go developers building an executable and easily shipping it to users? Wouldn't it be great if your users could **run your application without installing anything**? That is the dream, and PyInstaller is one way to get there in the Python ecosystem.

There are countless tutorials on how to set up virtual environments, manage dependencies, and publish to PyPI, which is useful when you're creating Python libraries. There is much less information for **developers building Python applications**. This tutorial is for developers who want to distribute applications to users who may or may not be Python developers.

All Tutorial Topics
advanced api basics best-practices community databases data-science devops django docker flask front-end gamedev gui intermediate machine-learning projects python testing tools web-dev web-scraping

Table of Contents
- Distribution Problems
- PyInstaller
- Preparing Your Project
- Using PyInstaller
- Digging Into PyInstaller Artifacts
- Customizing Your Builds
- Testing Your New Executable
- Debugging PyInstaller Executables
- Limitations
- Conclusion

Mark as Completed
Tweet Share Email

In this tutorial, you'll learn the following:

- How PyInstaller can simplify application distribution
- How to use PyInstaller on your own projects
- How to debug PyInstaller errors
- What PyInstaller can't do

PyInstaller gives you the ability to create a folder or executable that users can immediately run without any extra installation. To fully appreciate PyInstaller's power, it's useful to revisit some of the distribution problems PyInstaller helps you avoid.

Distribution Problems

Setting up a Python project can be frustrating, especially for non-developers. Often, the setup starts with opening a Terminal, which is a non-starter for a huge group of potential users. This roadblock stops users even before the installation guide delves into the complicated details of virtual environments, Python versions, and the myriad of potential dependencies.

Think about what you typically go through when setting up a new machine for Python development. It probably goes something like this:

- Download and install a specific version of Python
- Set up pip
- Set up a virtual environment
- Get a copy of your code
- Install dependencies

Stop for a moment and consider if any of the above steps make any sense if you're not a developer, let alone a Python developer. Probably not.

These problems explode if your user is lucky enough to get to the dependencies portion of the installation. This has gotten much better in the last few years with the prevalence of wheels, but some dependencies still require C/C++ or even FORTRAN compilers!

This barrier to entry is way too high if your goal is to make an application available to as many users as possible. As Raymond Hettinger often says in his excellent talks, "There has to be a better way."

PyInstaller

PyInstaller abstracts these details from the user by finding all your dependencies and bundling them together. Your users won't even know they're running a Python project because the Python Interpreter itself is bundled into your application. Goodbye complicated installation instructions!

PyInstaller performs this amazing feat by introspecting your Python code, detecting your dependencies, and then packaging them into a suitable format depending on your Operating System.

There are lots of interesting details about PyInstaller, but for now you'll learn the basics of how it works and how to use it. You can always refer to the excellent PyInstaller docs if you want more details.

In addition. PyInstaller can create executables for Windows. Linux. or macOS. This means

Windows users will get a `.exe`, Linux users get a regular executable, and macOS users get a `.app` bundle. There are some caveats to this. See the limitations section for more information.

Preparing Your Project

PyInstaller requires your application to conform to some minimal structure, namely that you have a CLI script to start your application. Often, this means creating a small script *outside* of your Python package that simply imports your package and runs `main()`.

The entry-point script is a Python script. You can technically do anything you want in the entry-point script, but you should avoid using explicit relative imports. You can still use relative imports throughout the rest your application if that's your preferred style.

> **Note:** An entry-point is the code that starts your project or application.

You can give this a try with your own project or follow along with the Real Python feed reader project. For more detailed information on the reader project, check out the the tutorial on Publishing a Package on PyPI.

The first step to building an executable version of this project is to add the entry-point script. Luckily, the feed reader project is well structured, so all you need is a short script *outside the package* to run it. For example, you can create a file called `cli.py` alongside the reader package with the following code:

Python

```
from reader.__main__ import main

if __name__ == '__main__':
    main()
```

This `cli.py` script calls `main()` to start up the feed reader.

Creating this entry-point script is straightforward when you're working on your own project because you're familiar with the code. However, it's not as easy to find the entry-point of another person's code. In this case, you can start by looking at the `setup.py` file in the third-party project.

Look for a reference to the `entry_points` argument in the project's `setup.py`. For example, here's the reader project's `setup.py`:

Python

```
setup(
    name="realpython-reader",
    version="1.0.0",
    description="Read the latest Real Python tutorials",
    long_description=README,
    long_description_content_type="text/markdown",
    url="https://github.com/realpython/reader",
    author="Real Python",
    author_email="info@realpython.com",
    license="MIT",
    classifiers=[
        "License :: OSI Approved :: MIT License",
        "Programming Language :: Python",
        "Programming Language :: Python :: 2",
        "Programming Language :: Python :: 3",
    ],
    packages=["reader"],
    include_package_data=True,
    install_requires=[
        "feedparser", "html2text", "importlib_resources", "typing"
    ],
    entry_points={"console_scripts": ["realpython=reader.__main__:main"]},
)
```

As you can see, the entry-point `cli.py` script calls the same function mentioned in the `entry_points` argument.

After this change, the reader project directory should look like this, assuming you checked it out into a folder called `reader`:

```
reader/
│
├── reader/
│   ├── __init__.py
│   ├── __main__.py
│   ├── config.cfg
│   ├── feed.py
│   └── viewer.py
│
├── cli.py
├── LICENSE
├── MANIFEST.in
├── README.md
├── setup.py
└── tests
```

Notice there is no change to the reader code itself, just a new file called `cli.py`. This entry-point script is usually all that's necessary to use your project with PyInstaller.

However, you'll also want to look out for uses of `__import__()` or imports inside of functions. These are referred to as hidden imports in PyInstaller terminology.

You can manually specify the hidden imports to force PyInstaller to include those dependencies if changing the imports in your application is too difficult. You'll see how to do this later in this tutorial.

Once you can launch your application with a Python script *outside* of your package, you're ready to give PyInstaller a try at creating an executable.

Using PyInstaller

The first step is to install PyInstaller from PyPI. You can do this using `pip` like other Python packages:

Shell

```
$ pip install pyinstaller
```

`pip` will install PyInstaller's dependencies along with a new command: `pyinstaller`. PyInstaller can be imported in your Python code and used as a library, but you'll likely only use it as a CLI tool.

You'll use the library interface if you create your own hook files.

You'll increase the likelihood of PyInstaller's defaults creating an executable if you only have pure Python dependencies. However, don't stress too much if you have more complicated dependencies with C/C++ extensions.

PyInstaller supports lots of popular packages like NumPy, PyQt, and Matplotlib without any additional work from you. You can see more about the list of packages that PyInstaller officially supports by referring to the PyInstaller documentation.

Don't worry if some of your dependencies aren't listed in the official docs. Many Python packages work fine. In fact, PyInstaller is popular enough that many projects have explanations on how to get things working with PyInstaller.

In short, the chances of your project working out of the box are high.

To try creating an executable with all the defaults, simply give PyInstaller the name of your main entry-point script.

First, `cd` in the folder with your entry-point and pass it as an argument to the `pyinstaller` command that was added to your `PATH` when PyInstaller was installed.

For example, type the following after you `cd` into the top-level `reader` directory if you're following along with the feed reader project:

Shell

```
$ pyinstaller cli.py
```

Don't be alarmed if you see a lot of output while building your executable. PyInstaller is verbose by default, and the verbosity can be cranked way up for debugging, which you'll see later.

Digging Into PyInstaller Artifacts

PyInstaller is complicated under the hood and will create a lot of output. So, it's important to know what to focus on first. Namely, the executable you can distribute to your users and potential debugging information. By default, the `pyinstaller` command will create a few things of interest:

- A `*.spec` file
- A `build/` folder
- A `dist/` folder

Spec File

The spec file will be named after your CLI script by default. Sticking with our previous example, you'll see a file called `cli.spec`. Here's what the default spec file looks like after running PyInstaller on the `cli.py` file:

Python

```
# -*- mode: python -*-

block_cipher = None

a = Analysis(['cli.py'],
             pathex=['/Users/realpython/pyinstaller/reader'],
             binaries=[],
             datas=[],
             hiddenimports=[],
             hookspath=[],
             runtime_hooks=[],
             excludes=[],
             win_no_prefer_redirects=False,
             win_private_assemblies=False,
             cipher=block_cipher,
             noarchive=False)
pyz = PYZ(a.pure, a.zipped_data,
             cipher=block_cipher)
exe = EXE(pyz,
          a.scripts,
          [],
          exclude_binaries=True,
          name='cli',
          debug=False,
          bootloader_ignore_signals=False,
          strip=False,
          upx=True,
          console=True )
coll = COLLECT(exe,
               a.binaries,
               a.zipfiles,
               a.datas,
               strip=False,
               upx=True,
               name='cli')
```

This file will be automatically created by the `pyinstaller` command. Your version will have different paths, but the majority should be the same

different paths, but the majority should be the same.

Don't worry, you don't need to understand the above code to effectively use PyInstaller!

This file can be modified and re-used to create executables later. You can make future builds a bit faster by providing this spec file instead of the entry-point script to the `pyinstaller` command.

There are a few specific use-cases for PyInstaller spec files. However, for simple projects, you won't need to worry about those details unless you want to heavily customize how your project is built.

Build Folder

The `build/` folder is where PyInstaller puts most of the metadata and internal bookkeeping for building your executable. The default contents will look something like this:

```
build/
│
└── cli/
    ├── Analysis-00.toc
    ├── base_library.zip
    ├── COLLECT-00.toc
    ├── EXE-00.toc
    ├── PKG-00.pkg
    ├── PKG-00.toc
    ├── PYZ-00.pyz
    ├── PYZ-00.toc
    ├── warn-cli.txt
    └── xref-cli.html
```

The build folder can be useful for debugging, but unless you have problems, this folder can largely be ignored. You'll learn more about debugging later in this tutorial.

Dist Folder

After building, you'll end up with a `dist/` folder similar to the following:

```
dist/
│
└── cli/
    └── cli
```

The `dist/` folder contains the final artifact you'll want to ship to your users. Inside the `dist/` folder, there is a folder named after your entry-point. So in this example, you'll have a `dist/cli` folder that contains all the dependencies and executable for our application. The executable to run is `dist/cli/cli` or `dist/cli/cli.exe` if you're on Windows.

You'll also find lots of files with the extension `.so`, `.pyd`, and `.dll` depending on your Operating System. These are the shared libraries that represent the dependencies of your project that PyInstaller created and collected.

> **Note:** You can add `*.spec`, `build/`, and `dist/` to your `.gitignore` file to keep `git status` clean if you're using `git` for version control. The default GitHub gitignore file for Python projects already does this for you.

You'll want to distribute the entire `dist/cli` folder, but you can rename `cli` to anything that suits you.

At this point you can try running the `dist/cli/cli` executable if you're following along with the feed reader example.

You'll notice that running the executable results in errors mentioning the `version.txt` file. This is because the feed reader and its dependencies require some extra data files that PyInstaller doesn't know about. To fix that, you'll have to tell PyInstaller that `version.txt` is required, which you'll learn about when testing your new executable.

Customizing Your Builds

PyInstaller comes with lots of options that can be provided as spec files or normal CLI options. Below, you'll find some of the most common and useful options.

`--name`

> Change the name of your executable.

This is a way to avoid your executable, spec file, and build artifact folders being named after your entry-point script. `--name` is useful if you have a habit of naming your entry-point script something like `cli.py`, as I do.

You can build an executable called `realpython` from the `cli.py` script with a command like this:

Shell

```
$ pyinstaller cli.py --name realpython
```

`--onefile`

> Package your entire application into a single executable file.

The default options create a folder of dependencies *and* and executable, whereas `--onefile` keeps distribution easier by creating *only* an executable.

This option takes no arguments. To bundle your project into a single file, you can build with a command like this:

Shell

```
$ pyinstaller cli.py --onefile
```

With the above command, your `dist/` folder will only contain a single executable instead of a folder with all the dependencies in separate files.

`--hidden-import`

> List multiple top-level imports that PyInstaller was unable to detect automatically.

This is one way to work around your code using `import` inside functions and `__import__()`. You can also use `--hidden-import` multiple times in the same command.

This option requires the name of the package that you want to include in your executable. For example, if your project imported the requests library inside of a function, then PyInstaller would not automatically include `requests` in your executable. You could use the following command to force `requests` to be included:

Shell

```
$ pyinstaller cli.py --hiddenimport=requests
```

You can specify this multiple times in your build command, once for each hidden import.

`--add-data` and `--add-binary`

> Instruct PyInstaller to insert additional data or binary files into your build.

This is useful when you want to bundle in configuration files, examples, or other non-code data. You'll see an example of this later if you're following along with the feed reader project.

`--exclude-module`

> Exclude some modules from being included with your executable

This is useful to exclude developer-only requirements like testing frameworks. This is a great way to keep the artifact you give users as small as possible. For example, if you use pytest, you may want to exclude this from your executable:

Shell

```
$ pyinstaller cli.py --exclude-module=pytest
```

`-w`

> Avoid automatically opening a console window for `stdout` logging.

This is only useful if you're building a GUI-enabled application. This helps your hide the details of your implementation by allowing users to never see a terminal.

Similar to the `--onefile` option, `-w` takes no arguments:

Shell

```
$ pyinstaller cli.py -w
```

`.spec file`

As mentioned earlier, you can reuse the automatically generated `.spec` file to further customize your executable. The `.spec` file is a regular Python script that implicitly uses the PyInstaller library API.

Since it's a regular Python script, you can do almost anything inside of it. You can refer to the official PyInstaller Spec file documentation for more information on that API.

Testing Your New Executable

The best way to test your new executable is on a new machine. The new machine should have the same OS as your build machine. Ideally, this machine should be as similar as possible to what your users use. That may not always be possible, so the next best thing is testing on your own machine.

The key is to run the resulting executable *without* your development environment activated. This means run without `virtualenv`, `conda`, or any other **environment** that can access your Python installation. Remember, one of the main goals of a PyInstaller-created executable is for users to not need anything installed on their machine.

Picking up with the feed reader example, you'll notice that running the default `cli` executable in the `dist/cli` folder fails. Luckily the error points you to the problem:

Shell

```
FileNotFoundError: 'version.txt' resource not found in 'importlib_resources'
[15110] Failed to execute script cli
```

The `importlib_resources` package requires a `version.txt` file. You can add this file to the build using the `--add-data` option. Here's an example of how to include the required `version.txt` file:

```
Shell
$ pyinstaller cli.py \
    --add-data venv/reader/lib/python3.6/site-packages/importlib_resources/version.txt
```

This command tells PyInstaller to include the `version.txt` file in the `importlib_resources` folder in a new folder in your build called `importlib_resources`.

> **Note:** The `pyinstaller` commands use the `\` character to make the command easier to read. You can omit the `\` when running commands on your own or copy and paste the commands as-is below provided you're using the same paths.

You'll want to adjust the path in the above command to match where you installed the feed reader dependencies.

Now running the new executable will result in a new error about a `config.cfg` file.

This file is required by the feed reader project, so you'll need to make sure to include it in your build:

```
Shell
$ pyinstaller cli.py \
    --add-data venv/reader/lib/python3.6/site-packages/importlib_resources/version.txt
    --add-data reader/config.cfg:reader
```

Again, you'll need to adjust the path to the file based on where you have the feed reader project.

At this point, you should have a working executable that can be given directly to users!

Debugging PyInstaller Executables

As you saw above, you might encounter problems when running your executable. Depending on the complexity of your project, the fixes could be as simple as including data files like the feed reader example. However, sometimes you need more debugging techniques.

Below are a few common strategies that are in no particular order. Often times one of these strategies or a combination will lead to a break-through in tough debugging sessions.

Use the Terminal

First, try running the executable from a terminal so you can see all the output.

Remember to remove the `-w` build flag to see all the `stdout` in a console window. Often, you'll see `ImportError` exceptions if a dependency is missing.

Debug Files

Inspect the `build/cli/warn-cli.txt` file for any problems. PyInstaller creates *lots* of output to help you understand exactly what it's creating. Digging around in the `build/` folder is a great place to start.

Single Directory Builds

Use the `--onedir` distribution mode of creating distribution folder instead of a single executable. Again, this is the default mode. Building with `--onedir` gives you the opportunity to inspect all the dependencies included instead of everything being hidden in a single executable.

`--onedir` is useful for debugging, but `--onefile` is typically easier for users to comprehend. After debugging you may want to switch to `--onefile` mode to simplify distribution.

Additional CLI Options

PyInstaller also has options to control the amount of information printed during the build process. Rebuild the executable with the `--log-level=DEBUG` option to PyInstaller and review the output.

PyInstaller will create *a lot* of output when increasing the verbosity with `--log-level=DEBUG`. It's useful to save this output to a file you can refer to later instead of scrolling in your Terminal. To do this, you can use your shell's redirection functionality. Here's an example:

Shell

```
$ pyinstaller --log-level=DEBUG cli.py 2> build.txt
```

By using the above command, you'll have a file called `build.txt` containing lots of additional `DEBUG` messages.

> **Note:** The standard redirection with `>` is not sufficient. PyInstaller prints to the `stderr` stream, *not* `stdout`. This means you need to redirect the `stderr` stream to a file, which can be done using a `2` as in the previous command.

Here's a sample of what your `build.txt` file might look like:

Text

```
67 INFO: PyInstaller: 3.4
67 INFO: Python: 3.6.6
73 INFO: Platform: Darwin-18.2.0-x86_64-i386-64bit
74 INFO: wrote /Users/realpython/pyinstaller/reader/cli.spec
74 DEBUG: Testing for UPX ...
77 INFO: UPX is not available.
78 DEBUG: script: /Users/realptyhon/pyinstaller/reader/cli.py
78 INFO: Extending PYTHONPATH with paths
['/Users/realpython/pyinstaller/reader',
 '/Users/realpython/pyinstaller/reader']
```

This file will have a lot of detailed information about what was included in your build, why something was not included, and how the executable was packaged.

You can also rebuild your executable using the `--debug` option in addition to using the `--log-level` option for even more information.

> **Note:** The `-y` and `--clean` options are useful when rebuilding, especially when initially configuring your builds or building with Continuous Integration. These options remove old builds and omit the need for user input during the build process.

Additional PyInstaller Docs

The PyInstaller GitHub Wiki has lots of useful links and debugging tips. Most notably are the sections on making sure everything is packaged correctly and what to do if things go wrong.

Assisting in Dependency Detection

The most common problem you'll see is `ImportError` exceptions if PyInstaller couldn't properly detect all your dependencies. As mentioned before, this can happen if you're using `__import__()`, imports inside functions, or other types of hidden imports.

Many of these types of problems can be resolved by using the `--hidden-import` PyInstaller

CLI option. This tells PyInstaller to include a module or package even if it doesn't automatically detect it. This is the easiest way to work around lots of dynamic import magic in your application.

Another way to work around problems is hook files. These files contain additional information to help PyInstaller package up a dependency. You can write your own hooks and tell PyInstaller to use them with the `--additional-hooks-dir` CLI option.

Hook files are how PyInstaller itself works internally so you can find lots of example hook files in the PyInstaller source code.

Limitations

PyInstaller is incredibly powerful, but it does have some limitations. Some of the limitations were discussed previously: hidden imports and relative imports in entry-point scripts.

PyInstaller supports making executables for Windows, Linux, and macOS, but it cannot cross compile. Therefore, you cannot make an executable targeting one Operating System from another Operating System. So, to distribute executables for multiple types of OS, you'll need a build machine for each supported OS.

Related to the cross compile limitation, it's useful to know that PyInstaller does not technically bundle absolutely everything your application needs to run. Your executable is still dependent on the users' `glibc`. Typically, you can work around the `glibc` limitation by building on the oldest version of each OS you intend to target.

For example, if you want to target a wide array of Linux machines, then you can build on an older version of CentOS. This will give you compatibility with most versions newer than the one you build on. This is the same strategy described in PEP 0513 and is what the PyPA recommends for building compatible wheels.

In fact, you might want to investigate using the PyPA's manylinux docker image for your Linux build environment. You could start with the base image then install PyInstaller along with all your dependencies and have a build image that supports most variants of Linux.

Conclusion

PyInstaller can help make complicated installation documents unnecessary. Instead, your users can simply run your executable to get started as quickly as possible. The PyInstaller workflow can be summed up by doing the following:

1. Create an entry-point script that calls your main function.
2. Install PyInstaller.
3. Run PyInstaller on your entry-point.
4. Test your new executable.
5. Ship your resulting `dist/` folder to users.

Your users don't have to know what version of Python you used or that your application uses Python at all!

```
>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email Address

Send Me Python Tricks »

About **Luke Lee**

Luke has professionally written software for applications ranging from Python desktop and web applications to embedded C drivers for Solid State Disks. He has also spoken at PyCon, PyTexas, PyArkansas, PyconDE, and meetup groups.

» More about Luke

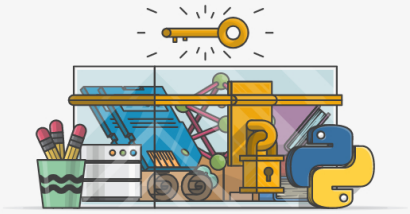
Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

 Aldren

 David

 Joanna

Master Real-World Python Skills With Unlimited Access to Real Python



Join us and get access to hundreds of tutorials, hands-on video courses, and a community of expert Pythonistas:

Level Up Your Python Skills »

What Do You Think?

Tweet Share Email

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won't make the cut here.

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Keep Learning

Related Tutorial Categories: best-practices intermediate