略有深度的话题(ADVANCED TOPICS)

在下面的内容中，我们会讨论一些PyInstaller内部实现方法的细节。对于大多数吃瓜群众而言，是没多大必要了解的。但如果你很好奇，想探索PyInstaller的内部代码，甚至想为此做出贡献，那不妨花一点时间看看此章节。

（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:

First process: bootloader starts.

If one-file mode, extract bundled files to temppath\_MEIxxxxxx

Modify various environment variables:

Linux: save original value of LD\_LIBRARY\_PATH into LD\_LIBRARY\_PATH\_ORIG, prepend our path to LD\_LIBRARY\_PATH.

AIX: same thing, but using LIBPATH and LIBPATH\_ORIG.

OSX: unset DYLD\_LIBRARY\_PATH.

Set up to handle signals for both processes.

Run the child process.

Wait for the child process to finish.

If one-file mode, delete temppath\_MEIxxxxxx.

Second process: bootloader itself started as a child process.

On Windows set the activation context.

Load the Python dynamic library. The name of the dynamic library is embedded in the executable file.

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

Run python code.

Running Python code requires several steps:

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.

Execute any run-time hooks: first those specified by the user, then any standard ones.

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.

Run the main script.

已打包好的应用中的Python导入内容(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:

Is it a built-in module? A list of built-in modules is in variable sys.builtin\_module\_names.

Is it a module embedded in the executable? Then load it from embedded archive.

Is it a C-extension? The app will try to find a file with name package.subpackage.module.pyd or package.subpackage.module.so

Next examine paths in the sys.path. There could be any additional location with python modules or .egg filenames.

If the module was not found then raise ImportError.

内容表与树形类(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.

内容表(TOC Classes)

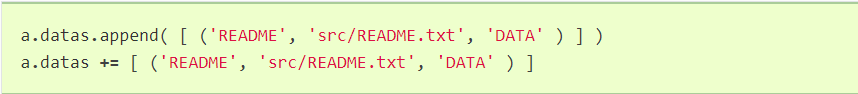
Objects of the TOC class are used as input to the classes created in a spec file. For example, the scripts member of an Analysis object is a TOC containing a list of scripts. The pure member is a TOC with a list of modules, and so on.

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

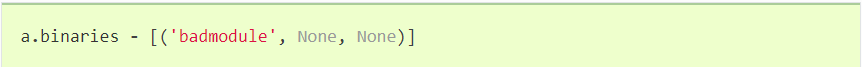
(name,path,typecode)

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:

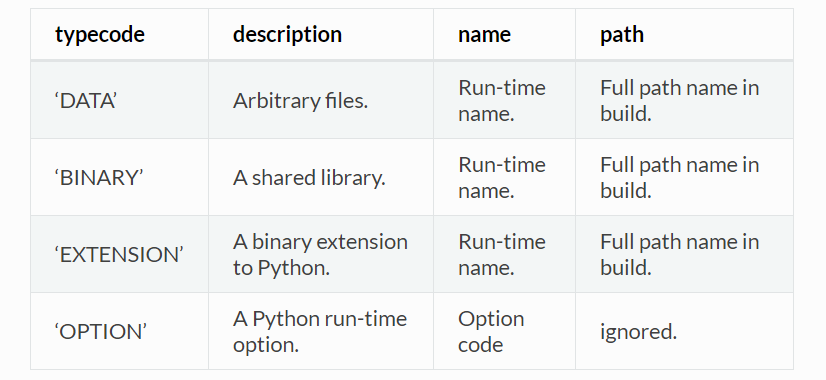


Set-difference makes excluding modules quite easy. For example:



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

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



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

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

树形类(The Tree Class)

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

Tree(root, prefix=run-time-folder, excludes=string\_list, typecode=code | 'DATA' )

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

The prefix argument, if given, is a name for a subfolder within the run-time folder to contain the tree files. If you omit prefix or give None, the tree files will be at the top level of the run-time folder.

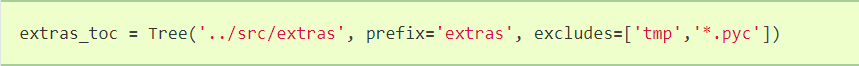
The excludes argument, if given, is a list of one or more strings that match files in the root that should be omitted from the Tree. An item in the list can be either:

a name, which causes files or folders with this basename to be excluded

\*.ext, which causes files with this extension to be excluded

The typecode argument, if given, specifies the TOC typecode string that applies to all items in the Tree. If omitted, the default is DATA, which is appropriate for most cases.

For example:



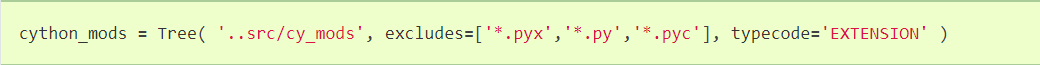
This creates extras\_toc as a TOC object that lists all files from the relative path ../src/extras, omitting those that have the basename (or are in a folder named) tmp or that have the type .pyc. Each tuple in this TOC has:

A name composed of extras/filename.

A path consisting of a complete, absolute path to that file in the ../src/extras folder (relative to the location of the spec file).

A typecode of DATA (by default).

An example of creating a TOC listing some binary modules:



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

Its own filename as name (no prefix; the file will be at the top level of the bundle).

A path as an absolute path to that file in ../src/cy\_mods relative to the spec file.

A typecode of EXTENSION (BINARY could be used as well).

检视压缩内容(Inspecting Archives)

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

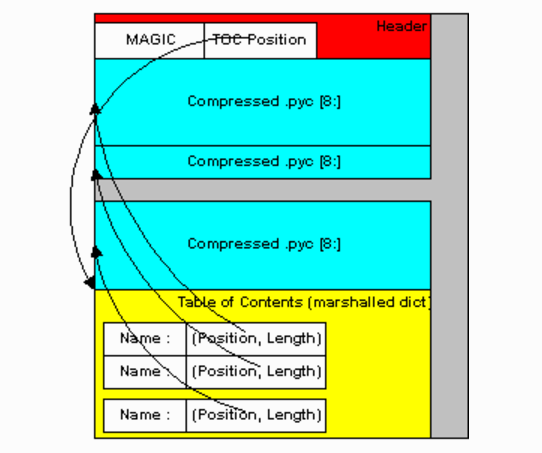
ZlibArchive

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

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

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

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



Carchive

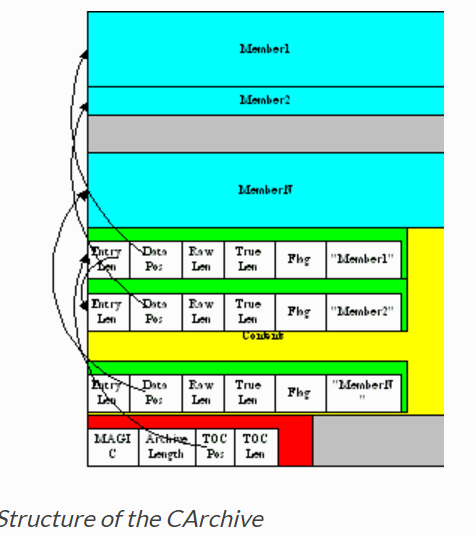
A CArchive can contain any kind of file. It’s very much like a .zip file. They are easy to create in Python and easy to unpack from C code. A CArchive can be appended to another file, such as an ELF and COFF executable. To allow this, the archive is made with its table of contents at the end of the file,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

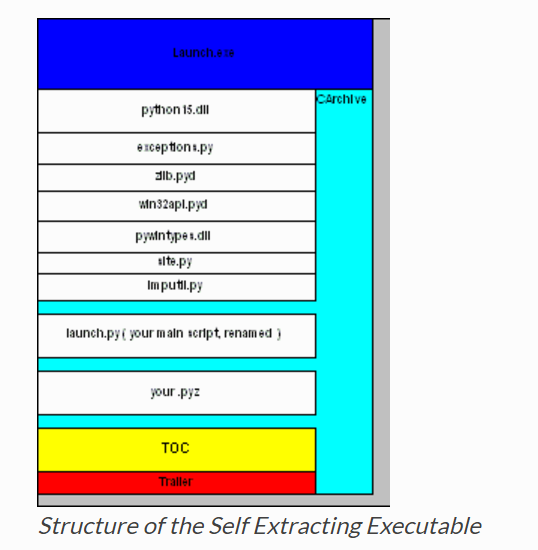
A CArchive can be embedded within another CArchive. An inner archive can be opened and used in place, without having to extract it.

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

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

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





Using pyi-archive\_viewer

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

pyi-archive\_viewer archivefile

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

O name

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

U

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

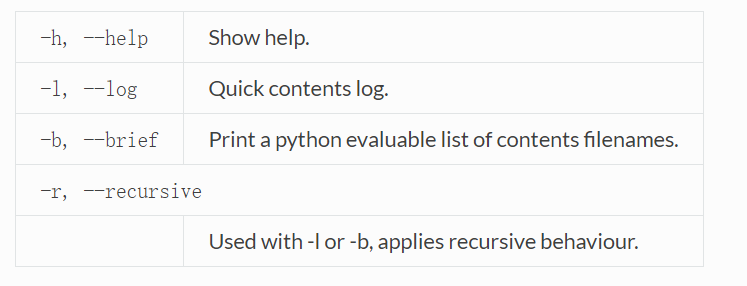
X name

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:



检视可执行文件(Inspecting Executables)

You can inspect any executable file with pyi-bindepend:

pyi-bindepend executable\_or\_dynamic\_library

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

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

Creating a Reproducible Build

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

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

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

