**Overview of Python Packaging**

As a general-purpose programming language, Python is designed to be used in many ways. You can build web sites or industrial robots or a game for your friends to play, and much more, all using the same core technology.

Python’s flexibility is why the first step in every Python project must be to think about the project and must be to think about the projects audience and the corresponding environment where the project will run. It might seem strange to think about packaging before writing code, but this process does wonders for avoiding future headaches.

This overview provides a general-purpose decision tree for reasoning about Python’s plethora of packaging options. Go through the “choose the best technology for your next project.”

**Thinking about deployment**

Packages exist to be installed (or deployed), so before you package anything, you’ll want to have some answers to the deployment below:

* Who are your software’s users? Will your software be installed by other developers doing software development, operations people in a datacenter, or less software-savvy group?
* Is your software intended to run on servers, desktops, mobile clients (phones, tables, etc), or embedded in dedicated devices?
* Is your software installed individually, or in large deployment batches?

Packaging is all about target environment and deployment experience. There are many answers to the questions above and each combination of circumstances has its own solutions. With this information, the following overview will guide you to the packaging technologies best suited to your project.

**Packaging Python libraries and tools**

You may have heard about PyPI, *setup.py* and wheel files. These are just a few of the tools Python’s ecosystem provides for distributing Python code to developers, which you can read about in [Packaging and distributing projects](https://packaging.python.org/guides/distributing-packages-using-setuptools/).

The following approaches to packaging are meant for libraries and tools used by technical audience in a development setting, skip ahead to [Packaging Python applications](https://packaging.python.org/overview/#packaging-applications).

**Python modules**

A Python file, provided it only relies on the standard library, can be redistributed and reused. You will also need to ensure it’s written for the right version of Python, and only relies on the standard library.

This is great for sharing simple scripts and snippets between people who both have compatible Python versions (such as via email, StackOverflow, or GitHub gists). There are even some entire Python libraries that offer this as an option, such as [bottle.py](https://bottlepy.org/docs/dev/) and [boltons](https://boltons.readthedocs.io/en/latest/architecture.html#architecture).

However, this pattern won’t scale for projects that consist of multiple files, need additional libraries, or need a specific version of Python, hence the options below.

**Python source distributions**

If your code consists of multiple Python files, it’s usually organized into a directory structure. Any directory containing Python files can comprise an [import package](https://packaging.python.org/glossary/#term-import-package).

Because packages consist of multiple files, they are harder to distribute. Most protocols support transferring only one file at a time. It’s easier to get incomplete transfers, and harder to guarantee code integrity at the destination.

So long as your code contains nothing but pure Python code, and you know your deployment environment supports your version of Python, then you can use Python’s native packaging tools to create a source distribution [package](https://packaging.python.org/glossary/#term-distribution-package), or sdist for short.

**Python binary distributions**

So much of Python's practical power comes from its ability to integrate with the software ecosystem, in particular libraries written in C, C++, Fortran, Rust, and other languages.

Not all developers have the right tools or experiences to build these components in these compiled languages, so Python created the wheel, a package format designed to ship libraries with compiled artifacts. In fact, Python’s package installer, pip, always prefers wheels because installation is always faster, so even pure Python packages work better with wheels.

Binary distributions are best when they come with source distributions to match. Even if you don’t upload wheels of your code for every operating system, by uploading the sdist, you’re enabling users of other platforms to still build it for themselves. Default to publishing both sdist and wheel archives together, unless you’re creating artifacts for a very specific use case where you know the recipient only needs one or the other.

Python and PyPI make it easy to upload both wheels and sdists together. Just follow the [Python Packaging](https://packaging.python.org/tutorials/packaging-projects/) Projects tutorial.

**Packaging Python applications.**

So far we’ve only discussed Python’s native distribution tools. Based on our introduction, you would be correct to infer these built-in approaches only target environments which have Python, and an audience who knows how to install Python packages.

With the variety of operating systems configurations, and people out there, this assumption is only safe when targeting a developer audience.

Python’s native packaging is mostly built for distribution reusable code, called libraries, between developers. You can piggyback tools, or basic applications for developers, on top of Python’s library packaging, using technologies like [setuptools entry points](http://setuptools.readthedocs.io/en/latest/setuptools.html#automatic-script-creation).

Libraries are building blocks, not complete applications. For distribution applications, there’s a whole new world of technologies out there.

The next few sections organize these application packaging options according to their dependencies on the target environment, so you can choose the right one for your project.

**Depending on a framewor**k

Some types of Python applications, like web site backends and other network services, are common enough that they have frameworks to enable their development and packaging. Other types of applications, like dynamic web front-ends and mobile clients, are complex enough to target that a framework becomes more than a convenience.

**Service platforms**

If you’re developing for “Platform-as-a-Service” or “PaaS” like Heroku or Google App Engine, you are going to want to follow their respective packaging guides.

* [Heroku](https://devcenter.heroku.com/articles/getting-started-with-python)
* [Google App Engine](https://cloud.google.com/appengine/docs/python/)
* [PythonAnywher](https://www.pythonanywhere.com/)e
* [OpenShift](https://blog.openshift.com/getting-started-python/)
* “Serverless” frameworks like [Zappa](https://github.com/Miserlou/Zappa)

In all these setups, the platform takes care of packaging and deployment, as long as you follow their patterns.

Most software does not fit one of these templates, hence the existence of all the other options below.

If you're developing software that will be deployed to machines you own, users’ personal computers, or any other arrangement, read on.

**Web browsers and mobile applications.**

Python’s steady advances are leading it into new spaces. These days you can write a mobile app or web application using Python. While the language may be familiar, the packaging and deployment practices are brand new.

If you’re planning on releasing to these new frontiers, you’ll want to check out the following frameworks, and refer to their packaging guides:

* [Kivy](https://kivy.org/#home)
* [Beeware](https://pybee.org/)
* [Brython](https://brython.info/)
* [Flexx](http://flexx.readthedocs.io/en/latest/)

If you are not interested in using a framework or platform, or just wonder about some of the technologies and techniques utilized by the frameworks above, continue reading below.

**Depending on a pre-installed Python**

Pick an arbitrary computer, and depending on the context, there’s a very good chance Python is already installed. Included by default in most Linux and Mac operating systems for many years now, you can reasonably depend on Python preexisting in your data-centers or on the personal machines of developers and scientists.

**Technologies which support this model**:

* [PEX](https://github.com/pantsbuild/pex#pex)  (Python Executable)
* [zipapp](https://docs.python.org/3/library/zipapp.html)  (does not help manage dependencies, requires Python 3.5+)
* [shiv](https://github.com/linkedin/shiv#shiv) (requires Python 3)

**Depending on a separate software distribution ecosystem**

For a long time many operating systems, including Mac and Windows, lacked built-in package management. Only recently did these OSes gain so-called “app stores”, but even those focus on consumer applications and offer little for developers.

Developers long sought remedies, and in this struggle, emerged with their own package management solutions such as [Homebrew](https://brew.sh/). The most relevant alternative for Python developers is a package ecosystem called [Anaconda](https://en.wikipedia.org/wiki/Anaconda_(Python_distribution)). Anaconda is built around Python and is increasingly common in academic, analytical, and other data-oriented environments, even making its way into [server-oriented environments](https://medium.com/paypal-engineering).

Instructions on building and publishing for the Anaconda ecosystem:

* [Building libraries and applications with conda](https://conda.io/docs/user-guide/tutorials/index.html)
* [Transitioning a native Python package to Anaconda](https://conda.io/docs/user-guide/tutorials/build-pkgs-skeleton.html)

A similar model involves installing an alternative Python distribution, but does not support arbitrary operating system-level packages:

* [Enthought Canopy](https://www.enthought.com/product/canopy/)
* [ActiveState ActivePython](https://www.activestate.com/activepython)
* [WinPython](http://winpython.github.io/)

**Bringing you own Python executable**

Computing as we know it is defined by the ability to execute programs. Every operating system natively supports one or more formats of program they can natively execute.

There are many techniques and technologies which turn your Python program into one of these formats, most of which involve embedding the Python interpreter and any other dependencies into a single executable file.

This approach, called freezing, offers wide compatibility and seamless user experience, though requires multiple technologies, and a good amount of effort.

A selection of Python freezers:

* [pyInstaller](http://www.pyinstaller.org/) - Cross-platform
* [cx\_Freeze](https://anthony-tuininga.github.io/cx_Freeze/) - Cross-platform
* [constructor](https://github.com/conda/constructor) - For command-line installers
* [py2exe](http://www.py2exe.org/) - Windows only
* [py2app](https://py2app.readthedocs.io/en/latest/) - Mac only
* [bbFreeze](https://pypi.org/project/bbfreeze) - Windows, Linux, Python 2 only
* [osnap](https://github.com/jamesabel/osnap) - Windows and Mac
* [pynsist](https://pypi.org/project/pynsist/) - Windows only

Most of the above imply single-user deployments. For multi-component server applications, see [Chef Omnibus](https://github.com/chef/omnibus#-omnibus).

**Bringing your own userspace**

An increasing number of operating systems - including Linux, Mac OS, and Windows - can be set up to run applications packaged as lightweight images, using a relatively modern arrangement often referred to as [operating-system-level virtualization](https://en.wikipedia.org/wiki/Operating-system-level_virtualization), or containerization.

These techniques are mostly Python agnostic, because they package the whole OS filesystems, not just Python or Python packages.

Adoption is most extensive among linux servers, where the technology originated and where the technologies below work best:

* [AppImage](https://appimage.org/)
* [Docker](https://www.fullstackpython.com/docker.html)
* [Flatpak](https://flatpak.org/)
* [Snapcraft](https://snapcraft.io/)

**Bringing your own kernel**

Most operating systems support some form of classical virtualization, running applications packaged as images containing a full operating system of their own. Running these virtual machines, or VMs, is a mature approach, widespread in data center environments.

These techniques are mostly reserved for larger scale deployments in data centers, though certain complex applications can benefit from this packaging. Technologies are Python agnostic, and include:

* [Vagrant](https://www.vagrantup.com/)
* [VHD](https://en.wikipedia.org/wiki/VHD_(file_format)), [AMI](https://en.wikipedia.org/wiki/Amazon_Machine_Image), and [other formats](https://docs.openstack.org/image-guide/image-formats.html)
* OpenStack - A cloud management system in Python, with extensive VM support

**Operating system packages**

As mentioned in [Depending on a separate software distribution ecosystem](https://packaging.python.org/overview/#depending-on-a-separate-ecosystem) above, some operating systems have package managers of their own. If you’re very sure of the operating system you’re targeting, you can depend directly on a format like [deb](https://en.wikipedia.org/wiki/Deb_(file_format)) (for debian, Ubuntu, etc.) or [RPM](https://en.wikipedia.org/wiki/RPM_Package_Manager) (For Red Hat, Fedora, etc.), and use that built-in package manager to take care of installation, and even deployment. You can even use FPM to generate both deb and RPM’s from the same source.

In most deployment pipelines, the OS package manager is just one piece of the puzzle.

**virtualenv**

[Virtualenvs](http://python-guide.readthedocs.io/en/latest/dev/virtualenvs/) have been an indispensable tool for multiple generations of Python developers, but are slowly fading from view, as they are being wrapped by higher-level tools. With packaging in particular, virtualenvs are used as primitives in the [dh-virtualenv](http://dh-virtualenv.readthedocs.io/en/1.0/tutorial.html) tool and [osnap](https://github.com/jamesabel/osnap), both of which wrap virtualenvs in a self-contained way.

For production deployments, do not rely on running pip install from the Internet into a virtualenv, as one might do in a development environment. The overview above is full of much better solutions

**Security**

The further down the gradient you come, the harder it gets to update components of your package. Everything is more tightly bound together.

For example, if a kernel security issue emerges, and you’re deploying containers, the host system’s kernel can be updated without requiring a new build on behalf of the application. If you deploy VM images, you'll need a new build. Whether or not this dynamic images makes one option more secure is still a bit of an old debate, going back to the still-unsettled matter of [static versus dynamic linking](https://www.google.com/search?channel=fs&q=static+vs+dynamic+linking).

**Wrap up**

Packaging in Python has a bit of a reputation for being a bumpy ride. This impression is mostly a by-product of Python’s versatility. Once you understand the natural boundaries between each packaging solution, you begin to realize that the varied landscape is a small price Python programmers pay for using one of the most balanced, flexible languages available.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**End**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**