The Joy of Packaging Documentation

Release 0.1

Assorted

Aug 06, 2018

Contents

1 Packaging 3

Scipy 2018 Tutorial

CHAPTER 1

Packaging

Packaging from start to finish for both PyPI and conda

Warning: The list of changes integrated in the tutorial after it was first given at the SciPy 2018 conference can be found in the *Tutorial Content Updates* document.

1.1 Topics

1.1.1 Tutorial Schedule

Outline

How are we spending our afternoon?

Agenda

- 0:00-0:20 Getting setup and overview of packaging
- 0:20-0:45 python packages: the setup.py file
- **Break**
- 1:00-1:30 Building and uploading to PyPI
- 1:30-2:00 Binaries and dependencies
- 2:00-2:45 Exercises
- **Break**
- 3:00-3:15 Conda-build overview
- 3:15-3:45 Exercise

- 3:45-4:00 conda-forge

0:00-00:10 Getting setup for this Tutorial

There is a repo for this tutorial here:

https://github.com/python-packaging-tutorial/python-packaging-tutorial

or:

http://bit.ly/JoyOfPackaging

And the materials are rendered as html here:

https://python-packaging-tutorial.github.io/python-packaging-tutorial/

(linked from the git repo)

Clone that repo now – so you can follow along.

```
git clone https://github.com/python-packaging-tutorial/python-packaging-tutorial.
git
```

0:10-00:20 Overview of packaging

Overview

- What is a package, anyway?
- Source/binary
- Wheel vs conda packages
- PyPI/anaconda.org

0:20-0:45 python packages: the setup.py file

Making a Python Package

- Python packages – what are they?
- The setup.py file
- Specifying requirements
- When and how to "pin" requirements
- Let's make a package!

0:45-1:00 Building and uploading to PyPI

Building and Uploading to PyPi

- Packaging Terminology 101
- Building and publishing a python distribution

1:00-1:10 Break

1:10-1:30 Exercises

- Prepare environment.
- Build source distribution and wheel.
- Publish artifacts on PyPI.

1:30-2:00 Binaries and dependencies

Binaries and Dependencies

- Why we build Python packages with native binaries: 1) **performance** and 2) **library integration**
- Different components of the binary build process and their role: *headers, libraries, compilers, linkers, build systems, system introspection tools, package managers*
- Basic requirements for binary compatibility: a) **C-runtime library compatibility** and b) **shared library compatibilty**
- Joyous tools: **scikit-build**'s role in coordinating components of the binary build process and **conda**'s role in resolving dependencies and creating compatible platform binaries

2:00-2:45 Exercises

- Build a Python package with a C++-based C-extension.
- Build a Python package with a Cython-based C-extension.
- Build a distributable Linux wheel package.

2:45-3:00 Break

3:00-3:15 Conda-build overview

Conda Packages

3:15-3:30 Exercises

- Write a conda recipe for the sample package from previous exercises (pure python)
- noarch packages
- Upload to anaconda cloud

3:30-3:45 Exercises

- Recipe for package with compiled extensions
- Add compiled extension (source will be provided to students) to sample package
- Modify recipe, if needed

- Rebuild the package
- Version pinning (python, numpy)
- Split packages - multi-ecosystem ones
- Compiler packages + pin_downstream
- Interoperation with scikit-build

3:45-4:00 Automated building with cloud-based CI services

conda-forge (optional; as time allows)

CI service overview & Conda-forge – what are the pieces and how do they fit together?

- Recipe format
- staged-recipes
- feedstocks
- Re-rendering and conda-smithy
- Updating package when new version released
- Future direction/community needs
- Invitation to sprints
- Contributing to Conda-forge
- Intro to conda-forge: staged-recipes, maintainer role, contributing to an existing package
- conda-smithy lint/rerender
- Example to go from the conda-skeleton to a PR on staged-recipes
- Comment on some special cases: cython extensions, non-python pkgs, the use of the CIs, etc.
- Exercise: put a package on staged-recipes

1.1.2 Overview

Packages

What is a "package"?

- In a broad sense, anything you install using your package manager
- Some kinds of packages have implied behavior and requirements
- Unfortunate overloading: python "package": a folder that python imports

Package Managers and Repos

- Many package managers: some OS specific:
  - apt, yum, dnf, chocolatey, homebrew, etc.
- Some language specific:

  - NPM, pip, RubyGems
- And there are many online repositories of packages:
  - PyPI, anaconda.org, CRAN, CPAN

But they all contain:

- Some form of dependency management
- Artifact and/or source repository

The idea is that you install something, and have it *just work*.

Package types:

A package can be essentially in two forms:

- source
- binary

Focusing now on the Python world:

As Python is a dynamic language, this distinction can get a bit blurred:

There is little difference between a source and binary package *for a pure python package*

But if there is any compiled code in there, building from source can be a challenge:

- Binary packages are very helpful

Source Packages

A source package is all the source code required to build the package.

Package managers (like pip) can automatically build your package from source.

But:

- Your system needs the correct tools installed, compilers, build tools, etc
- You need to have the dependencies available
- Sometimes it takes time, sometimes a LONG time

Binary Packages

A collection of code all ready to run.

- Everything is already compiled and ready to go – makes it easy.

But:

- It's likely to be platform dependent
- May require dependencies to be installed

How do you manage that if the dependencies aren't in the same language/system?

Python Packaging

There are two package managers widely used for Python.

pip: The "official" solution.

- Pulls packages from PyPI
- Handles both source and binary packages (wheels)
- Python only

conda: Widely used in the scipy community.

- Pulls packages from anaconda.org
- Binary only (does not compile code when installing)
- Supports other languages / libraries: C, Fortran, R, Perl, Java (anything, really)
- Manages Python itself!

OS package managers:

- Linux: apt, conda, dnf, homebrew, nix, pacman, spack, yum
- OS-X: conda, homebrew, macports, spack
- Windows: chocolatey, conda, cygwin, pacman (msys2)

Sometimes handle python packages – but we won't talk about those here.

1.1.3 Making a Python Package

Specifying how to build your python package

Python Packages

What is a "package" in Python ?

Packages, modules, imports, oh my!

Modules

A python "module" is a single namespace, with a collection of values:

- functions
- constants
- class definitions
- really any old value.

A module usually corresponds to a single file: `something.py`

Packages

A "package" is essentially a module, except it can have other modules (and indeed other packages) inside it.

A package usually corresponds to a directory with a file in it called `__init__.py` and any number of python files or other package directories:

```
a_package
   __init__.py
   module_a.py
   a_sub_package
     __init__.py
     module_b.py
```

The `__init__.py` can be totally empty – or it can have arbitrary python code in it.

The code will be run when the package is imported – just like a module,

modules inside packages are *not* automatically imported. So, with the above structure:

```
import a_package
```

will run the code in `a_package/__init__.py`.

Any names defined in the `__init__.py` will be available in:

```
a_package.a_name
```

but:

```
a_package.module_a
```

will not exist. To get submodules, you need to explicitly import them:

`import a_package.module_a`

https://docs.python.org/3/tutorial/modules.html#packages

The module search path

The interpreter keeps a list of all the places that it looks for modules or packages when you do an import:

```
import sys
for p in sys.path:
    print p
```

You can manipulate that list to add or remove paths to let python find modules on a new place.

And every module has a `__file__` name that points to the path it lives in. This lets you add paths relative to where you are, etc.

NOTE: it's usually better to use setuptools' "develop" mode instead – see below.

Building Your Own Package

The very basics of what you need to know to make your own package.

Why Build a Package?

There are a bunch of nifty tools that help you build, install and distribute packages.

Using a well structured, standard layout for your package makes it easy to use those tools.

Even if you never want to give anyone else your code, a well structured package simplifies development.

What is a Package?

A collection of modules

. . . and the documentation

. . . and the tests

. . . and any top-level scripts

. . . and any data files required

. . . and a way to build and install it. . .

Python packaging tools:

`distutils`: included with python

```
from distutils.core import setup
```

Getting clunky, hard to extend, maybe destined for deprecation . . .

`setuptools`: for extra features, technically third party

- present in most modern Python installations

"The Python Packaging Authority" – PyPA

https://www.pypa.io/en/latest/

setuptools

`setuptools` is an extension to `distutils` that provides a number of extensions:

```
from setuptools import setup
```

superset of the `distutils setup`

This buys you a bunch of additional functionality:

- auto-finding packages
- better script installation
- resource (non-code files) management
- **develop mode**
- a LOT more

http://pythonhosted.org//setuptools/

Where do I go to figure this out?

This is a really good guide:

Python Packaging User Guide:

https://packaging.python.org/

and a more detailed tutorial:

http://python-packaging.readthedocs.io/en/latest/

Follow one of them

There is a sample project here:

https://github.com/pypa/sampleproject

(this has all the complexity you might need. . .)

You can use this as a template for your own packages.

Here is an opinionated update – a little more fancy, but some good ideas:

https://blog.ionelmc.ro/2014/05/25/python-packaging/

Basic Package Structure:

```
package_name/
    bin/
    CHANGES.txt
    docs/
    LICENSE.txt
    MANIFEST.in
    README.txt
    setup.py
    package_name/
          __init__.py
          module1.py
          module2.py
          test/
              __init__.py
              test_module1.py
              test_module2.py
```

`CHANGES.txt`: log of changes with each release

`LICENSE.txt`: text of the license you choose (do choose one!)

`MANIFEST.in`: description of what non-code files to include

`README.txt`: description of the package – should be written in ReST or Markdown (for PyPi):

`setup.py`: the script for building/installing package.

`bin/`: This is where you put top-level scripts

(some folks use `scripts`)

`docs/`: the documentation

`package_name/`: The main package – this is where the code goes.

`test/`: your unit tests. Options here:

Put it inside the package – supports

```
$ pip install package_name
>> import package_name.test
>> package_name.test.runall()
```

Or keep it at the top level.

Some notes on that:

‘ Where to put Tests <http://pythonchb.github.io/PythonTopics/where_to_put_tests.html>‘_

The `setup.py` File

Your `setup.py` file is what describes your package, and tells setuptools how to package, build and install it

It is python code, so you can add anything custom you need to it

But in the simple case, it is essentially declarative.

http://docs.python.org/3/distutils/

What Does `setup.py` Do?

- Version & package metadata
- List of packages to include
- List of other files to include
- List of dependencies
- List of extensions to be compiled (if you are not using scikit-build.

An example `setup.py`:

```
from setuptools import setup

setup(
  name='PackageName',
  version='0.1.0',
  author='An Awesome Coder',
  author_email='aac@example.com',
  packages=['package_name', 'package_name.test'],
  scripts=['bin/script1','bin/script2'],
  url='http://pypi.python.org/pypi/PackageName/',
  license='LICENSE.txt',
  description='An awesome package that does something',
  long_description=open('README.txt').read(),
  install_requires=[
      "Django >= 1.1.1",
      "pytest",
  ],
)
```

setup.cfg

Provides a way to give the end user some ability to customize the install

It's an `ini` style file:

```
[command]
option=value
...
```

simple to read and write.

`command` is one of the Distutils commands (e.g. build_py, install)

`option` is one of the options that command supports.

Note that an option spelled `--foo-bar` on the command-line is spelled `foo_bar` in configuration files.

Running setup.py

With a `setup.py` script defined, setuptools can do a lot:

Builds a source distribution (a tar archive of all the files needed to build and install the package):

```
python setup.py sdist
```

Builds wheels:

```
./setup.py bdist_wheel
```

(you need the wheel package for this to work:)

```
pip install wheel
```

Build from source:

```
python setup.py build
```

And install:

```
python setup.py install
```

Develop mode

Install in "develop" or "editable" mode:

```
python setup.py develop
```

or:

```
pip install .
```

Under Development

Develop mode is *really*, *really* nice:

```
$ python setup.py develop
```

or:

```
$ pip install -e ./
```

(the e stands for "editable" – it is the same thing)

It puts a link (actually `*.pth` files) into the python installation to your code, so that your package is installed, but any changes will immediately take effect.

This way all your test code, and client code, etc, can all import your package the usual way.

No `sys.path` hacking

Good idea to use it for anything more than a single file project.

Install	Development Install
Copies package into site-packages	Adds a `.pth` file to site-packages, pointed at package source root
Used when creating conda packages	Used when developing software locally
Normal priority in sys.path	End of `sys.path` (only found if nothing else comes first)

https://grahamwideman.wikispaces.com/Python-+site-package+dirs+and+.pth+files

Aside on pip and dependencies

- `pip` does not currently have a solver: http://github.com/pypa/pip/issues/988
- pip may replace packages in your environment with incompatible versions. Things will break when that happens.
- use caution (and ideally, disposable environments) when using pip

Getting Started With a New Package

For anything but a single-file script (and maybe even then):

1. Create the basic package structure
2. Write a `setup.py`
3. `pip install -e .`
4. Put some tests in `package/test`
5. `pytest` in the test dir, or `pytest --pyargs package_name`

or use "Cookie Cutter":

https://cookiecutter.readthedocs.io/en/latest/

Exercise: A Small Example Package

- Create a small package
  - package structure
  - `setup.py`
  - `python setup.py develop`
  - `at least one working test`

Start with the silly code in the tutorial repo in:

`python-packaging-tutorial/setup_example/`

or you can download a zip file here:

`capitalize.zip`

capitalize

capitalize is a useless little utility that will capitalize the words in a text file.

But it has the core structure of a python package:

- a library of "logic code"
- a command line script
- a data file
- tests

So let's see what's in there:

```
$ ls
capital_mod.py          test_capital_mod.py
cap_data.txt            main.py
cap_script.py           sample_text_file.txt
```

What are these files?

`capital_mod.py` The core logic code

`main.py` The command line app

`test_capital_mod.py` Test code for the logic

`cap_script.py` top-level script

`cap_data.txt` data file

`sample_text_file.txt` sample example file to test with.

Try it out:

```
$ cd capitalize/

$ python3 cap_script.py sample_text_file.txt
```

(continues on next page)

(continued from previous page)

```
Capitalizing: sample_text_file.txt and storing it in
sample_text_file_cap.txt

I'm done
```

So it works, as long as you are in the directory with all the code.

Setting up a package structure

Create a basic package structure:

```
package_name/
    bin/
    README.txt
    setup.py
    package_name/
          __init__.py
          module1.py
          test/
              __init__.py
              test_module1.py
```

Let's create all that for capitalize:

Make the package:

```
$ mkdir capitalize

$ cd capitalize/

$ touch __init__.py
```

Move the code into it:

```
    $ mv ../capital_mod.py ./
$ mv ../main.py ./
```

Create a dir for the tests:

```
$ mkdir test
```

Move the tests into that:

```
$ mv ../test_capital_mod.py test/
```

Create a dir for the script:

```
$ mkdir bin
```

Move the script into that:

```
$ mv ../cap_script.py bin
```

Create directory for data:

```
$ mkdir data
```

Move data into that:

```
$ mv ../cap_data.txt data
```

Now we have a package!

Let's try it:

```
$ python bin/cap_script.py
Traceback (most recent call last):
  File "bin/cap_script.py", line 8, in <module>
    import capital_mod
ImportError: No module named capital_mod
```

OK, that didn't work. Why not?

Well, we've moved everytihng around:

The modules don't know how to find each other.

Let's Write a `setup.py`

```
#!/usr/bin/env python

from setuptools import setup

setup(name='capitalize',
      version='1.0',
      # list folders, not files
      packages=['capitalize',
                'capitalize.test'],
      scripts=['capitalize/bin/cap_script.py'],
      package_data={'capitalize': ['data/cap_data.txt']},
      )
```

(remember that a "package" is a folder with a `__init__.py__` file)

That's about the minimum you can do.

Save it as `setup.py` *outside* the capitalize package dir.

Install it in "editable" mode:

```
$ pip install -e ./
Obtaining file:///Users/chris.barker/HAZMAT/Conferences/SciPy-2018/PackagingTutorial/
↪TutorialDay/capitalize
Installing collected packages: capitalize
  Running setup.py develop for capitalize
Successfully installed capitalize
```

Try it out:

```
$ cap_script.py
Traceback (most recent call last):
  File "/Users/chris.barker/miniconda2/envs/py3/bin/cap_script.py", line 6, in
↪<module>
```

(continues on next page)

(continued from previous page)

```
    exec(compile(open(__file__).read(), __file__, 'exec'))
  File "/Users/chris.barker/HAZMAT/Conferences/SciPy-2018/PackagingTutorial/
↪TutorialDay/capitalize/capitalize/bin/cap_script.py", line 8, in <module>
    import capital_mod
ModuleNotFoundError: No module named 'capital_mod'
```

Still didn't work – why not?

We need to update some imports.

in cap_script.py:

```
import main
import capital_mod
```

should be:

```
from capitalize import main
from capitalize import capital_mod
```

and similarly in main.py:

```
from capitalize import capital_mod
```

And try it:

```
$ cap_script.py sample_text_file.txt

      Traceback (most recent call last):
File ".../cap_script.py", line 6, in <module>
  exec(compile(open(__file__).read(), __file__, 'exec'))
File ".../cap_script.py", line 8, in <module>
  from capitalize import capital_mod
File "/.../capital_mod.py", line 35, in <module>
  special_words = load_special_words(get_datafile_name())
File ".../capital_mod.py", line 21, in load_special_words
  with open(data_file_name) as data_file:
FileNotFoundError: [Errno 2] No such file or directory: '.../capitalize/cap_data.txt'
```

Our script cannot find the data file. We changed it's location but not the path in the *capital_mod.py*.

Let's fix this. On line 32 replace:

```
return Path(__file__).parent / "cap_data.txt"
```

with:

```
return Path(__file__).parent / "data/cap_data.txt"
```

Running the tests:

Option 1: cd to the test dir:

```
$ cd capitalize/test/
```

(continues on next page)

(continued from previous page)

```
$ pytest
$ ==================================
  test session starts
  ==================================
...

Traceback:
test_capital_mod.py:14: in <module>
    import capital_mod
E   ModuleNotFoundError: No module named 'capital_mod'
```

Whoops – we need to fix that import, too:

```
from capitalize import capital_mod
```

And now we're good:

```
$ pytest
======test session starts =====

collected 3 items

test_capital_mod.py ...

============== 3 passed in 0.06 seconds ============
```

You can also run the tests from anywhere on the command line:

```
$ pytest --pyargs capitalize

    collected 3 items

    capitalize/capitalize/test/test_capital_mod.py ...                    ␣
↪    [100%]

    =============== 3 passed in 0.03 seconds ==========
```

Making Packages the Easy Way

To auto-build a full package structure:

Rather than doing it by hand, you can use the nifty "cookie cutter" project:

https://cookiecutter.readthedocs.io/en/latest/

And there are a few templates that can be used with that.

The core template written by the author:

https://github.com/audreyr/cookiecutter-pypackage

And one written by the author of the opinionated blog post above:

https://github.com/ionelmc/cookiecutter-pylibrary

Either are great starting points.

```
conda install -c conda-forge cookiecutter
```

or

```
pip install  cookiecutter
```

No time for that now :-(

Handling Requirements

Only the simplest of packages need only the Python standard library.

Requirements in `setup.py`

```
#!/usr/bin/env python
from distutils.core import setup

setup(name='mypkg',
      version='1.0',
      # list folders, not files
      packages=['mypkg', 'mypkg.subpkg'],
      install_requires=['click'],
      )
```

Requirements in `requirements.txt`

Common Mistake:

- requirements.txt often from pip freeze
- Pinned way too tightly. OK for env creation, bad for packaging.

- Donald Stufft (PyPA): Abstract vs. Concrete dependencies

Requirements in `setup.cfg` (ideal)

```
[metadata]
name = my_package
version = attr:
src.VERSION

[options]
packages = find:
install_requires = click
```

Parse-able without execution, unlike `setup.py`

configuring setup using setup cfg files

Break time!

Up next: producing redistributable artifacts

1.1.4 Building and Uploading to PyPi

Learning Objectives

In the following section we will . . .

- Review the packaging terminology
- Understand how to build, package and publish a python package

Packaging Terminology 101

Introduction

This section reviews the key python packaging concepts and definitions.

PyPI

PyPI is the default Package Index for the Python community. It is open to all Python developers to consume and distribute their **distributions**.

There are two instances of the Package Index:

- PyPI: Python Package Index hosted at https://pypi.org/
- TestPyPI: a separate instance of the Python Package Index (PyPI) that allows you to try out the distribution tools and process without worrying about affecting the real index. TestPyPI is hosted at https://test.pypi.org

Reference: https://packaging.python.org/glossary/#term-python-package-index-pypi

pip

The PyPA recommended tool for installing Python packages.

A multi-faceted tool:

- It is an *integration frontend* that takes a set of package requirements (e.g. a requirements.txt file) and attempts to update a working environment to satisfy those requirements. This may require locating, building, and installing a combination of **distributions**.
- It is a **build frontend** that can takes arbitrary source trees or source distributions and builds wheels from them.

Reference: http://pip.readthedocs.io/

PyPA

The Python Packaging Authority (PyPA) is a working group that maintains many of the relevant projects in Python packaging.

The associated website https://www.pypa.io references the PyPA Goals, Specifications and Roadmap as well as Python Packaging User Guide, a collection of tutorials and references to help you distribute and install Python packages with modern tools.

Reference: https://www.pypa.io

Source distribution

- Synonyms: sdist, Source release
- provides metadata + source files
- needed for installing
  - by a tool like pip
  - or for generating a Built Distribution

Reference: https://packaging.python.org/glossary/#term-source-distribution-or-sdist

Built Distribution

- Synonyms: bdist
- provides metadata + pre-built files
- only need to be moved (usually by pip) to the correct locations on the target system

Reference: https://packaging.python.org/glossary/#term-built-distribution

Python Distribution: pure vs non-pure

- **pure**:
  - Not specific to a CPU architecture
  - No ABI (Application Binary Interface)
- **non-pure**
  - ABI
  - Platform specific

Reference: https://packaging.python.org/glossary/#term-module

Binary Distribution

- is a **Built Distribution**
- is **non-pure**
- uses platform-specific compiled extensions

Reference: https://packaging.python.org/glossary/#term-binary-distribution

Wheel

- a **Built Distribution**
- a ZIP-format archive with .whl extension
  - `{distribution}-{version}(-{build tag})?-{python tag}-{abi tag}-{platform tag}.whl`
- described by PEP 427

Reference: https://packaging.python.org/glossary/#term-wheel

Wheels vs. Conda packages

Wheels	Conda packages
Employed by pip, blessed by PyPA	Foundation of Anaconda ecosystem
Used by any python installation	Used by conda python installations
Mostly specific to Python ecosystem	General purpose (any ecosystem)
Good mechanism for specifying range of python compatibility	Primitive support for multiple python versions (noarch)
Depends on static linking or other system package managers to provide core libraries	Can bundle core system-level shared libraries as packages, and resolve dependencies

To learn more about Conda, see *Conda Packages* section.

Virtual Environment

An isolated Python environment that allows packages to be installed for use by a particular application, rather than being installed system wide.

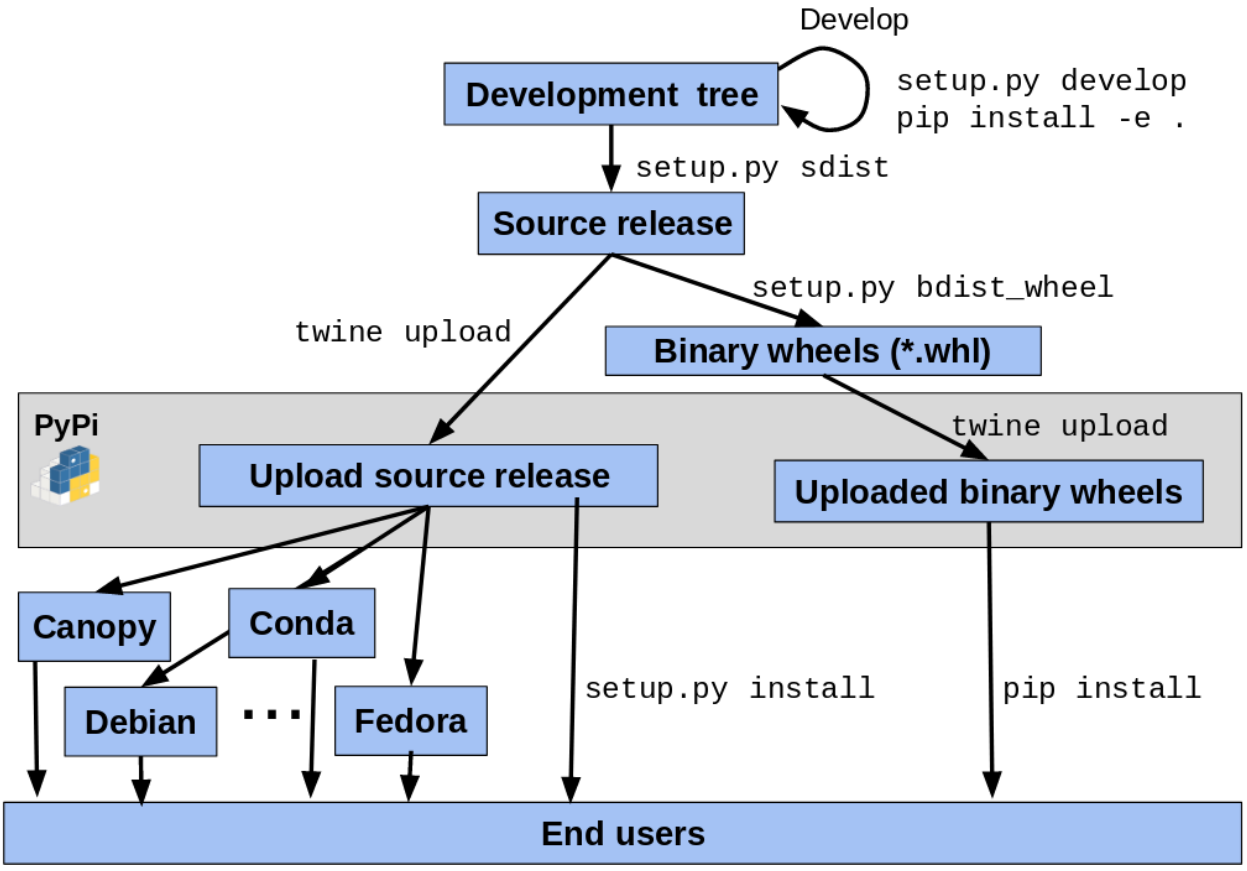
Learn more reading Creating Virtual Environments

Build system

Synonym: Build backend

- setuptools associated with the wheel package form the default build system. They support the creation of source and **built distributions** based on a `setup.py` and optionally a `setup.cfg` file.
- flit is an alternative backend allowing to also create (and also publish) **built distributions**.

Python Package Lifecycle



Tutorial

Introduction

This section discusses how to build python packages (or distributions) and publish them in a central repository to streamline their installation. Finally, we conclude with exercises where we publish a package with the Test Python Package Index.

Creating an environment

Before developing or building your distribution, we highly recommend to create a dedicated environment. This is supported by both `conda` and `pip`.

Building a source distribution

By leveraging the `setup.py` script, setuptools can build a source distribution (a tar archive of all the files needed to build and install the package):

```
$ python setup.py sdist

$ ls -1 dist
SomePackage-1.0.tar.gz
```

Building a wheel

```
$ pip wheel . -w dist

$ ls -1 dist
SomePackage-1.0-py2.py3-none-any.whl
```

This is equivalent to:

```
$ python setup.py bdist_wheel
```

Installing a wheel

- Install a package from PyPI:

```
$ pip install SomePackage
[...]
Successfully installed SomePackage
```

- Install a package from TestPyPI:

```
$ pip install -i https://test.pypi.org/simple SomePackage
[...]
Successfully installed SomePackage
```

- Install a package file:

```
$ pip install SomePackage-1.0-py2.py3-none-any.whl
[...]
Successfully installed SomePackage
```

For more details, see QuickStart guide from pip documentation.

Installing a source distribution

```
$ pip install SomePackage-1.0.tar.gz
    [...]
Successfully installed SomePackage
```

It transparently builds the associated wheel and install it.

Publishing to PyPI

twine utility is used for publishing Python packages on PyPI.

It is available as both a conda and a pypi package.

Learn more reading Using TestPiPY.

Exercises

Exercise 1: Prepare environment

- In the context of this tutorial, because participants already installed miniconda, we will create a conda environment and install packages using `conda install SomePackage`.

```
# create and activate a dedicated environment named "hello-pypi"
conda create -n hello-pypi -y -c conda-forge
conda activate hello-pypi

# install pip, wheel and twine
conda install -y twine wheel pip
```

- Create an account on TestPyPI (https://test.pypi.org/account/register/)

Exercise 2: Build source distribution and wheel

- Download (or checkout using git) the sources of our `hello-pypi` sample project:

```
conda install -y wget
wget https://github.com/python-packaging-tutorial/hello-pypi/archive/master.zip
```

- Extract sources

```
conda install -y unzip
unzip master.zip
cd hello-pypi-master
```

- Modify package name so that it is unique
- Then, build the source distribution:

```
$ python setup.py sdist
```

- And finally, build the wheel:

```
$ pip wheel . -w dist
```

- Make sure artifacts have been generated in the `dist` subdirectory.

Exercise 3: Publish artifacts on PyPI

```
$ twine upload --repository-url https://test.pypi.org/legacy/ dist/*
```

Bonus Exercise 4: Publish artifacts automating authentication

- Delete `hello-pypi-master` directory and extract archive again.
- Update name of package and rebuild source distribution and wheel.

- Create file `.pypirc` in your home directory with the following content:

```
[distutils]
index-servers=
    pypi
    testpypi

[testpypi]
repository: https://test.pypi.org/legacy/
username: your testpypi username
password: your testpypi password

[pypi]
username: your testpypi username
password: your testpypi password
```

- Publish package on TestPyPI:

```
$ twine upload --repository testpypi dist/*
```

Omitting the `-repository testpypi` argument allows to upload to the regular PyPI server.

Bonus Exercise 5: Setting up continuous integration

- See branch master-with-ci branch associated with `hello-pypi` example.

Resources

Where do I go to figure this out?

This is a really good guide:

Python Packaging User Guide:

https://packaging.python.org/

and a more detailed tutorial:

http://python-packaging.readthedocs.io/en/latest/

Follow one of them

There is a sample project here:

https://github.com/pypa/sampleproject

(this has all the complexity you might need. . .)

You can use this as a template for your own packages.

Here is an opinionated update – a little more fancy, but some good ideas:

https://blog.ionelmc.ro/2014/05/25/python-packaging/

Rather than doing it by hand, you can use the nifty "cookie cutter" project:

https://cookiecutter.readthedocs.io/en/latest/

And there are a few templates that can be used with that.

The core template written by the author:

https://github.com/audreyr/cookiecutter-pypackage

And one written by the author of the opinionated blog post above:

https://github.com/ionelmc/cookiecutter-pylibrary

Either are great starting points.

1.1.5 Binaries and Dependencies

Learning Objectives

In this section we will . . .

- Understand why we build Python packages with native binaries: 1) **performance** and 2) **library integration**
- Understand different components of the binary build process and their role: *headers, libraries, compilers, linkers, build systems, system introspection tools, package managers*
- Understand basic requirements for binary compatibility: a) **C-runtime library compatibility** and b) **shared library compatibilty**
- Understand **scikit-build**'s role in coordinating components of the binary build process and **conda**'s role in resolving dependencies and creating compatible platform binaries

Tutorial

Introduction

This section discusses the creation of Python packages that contain **native binaries**.

First, we explain why building Python packages with native binaries is often *desirable* or *necessary* for *scientific applications*.

Next, an overview of the requirements to build native binaries is provided. Within this the context, we explain how *scikit-build* and *conda-build* make life easier when we want to satisfy these requirements.

Finally, run an exercise where we build a native Python wth native binaries package and analyze the different stages of the build process.

Motivation

Scientific computing applications demand **higher performance** than other domains because of the:

1. **Size** of the **datasets** to be analyzed
2. **Complexity** of the **algorithms** evaluated

In order to achieve **high performance**, programs can:

1. **Minimized the number of operations** on the CPU required to acheive a certain task
2. **Execute in parallel** to leverage multi-core, many-core, and GPGPU system architectures
3. Carefully and precisely **manage memory** allocation and use

Greater performance is achieved with native binaries over CPython because:

1. Tasks are **compiled down to minimal processor operations**, as opposed to high level programming language instructions that must be **interpreted**
2. Parallel computing is not impared by CPython's Global Interpreter Lock (GIL)
3. **Memory** can be managed **explicitly** and **deterministically**

Many existing scientific codes are written in **programming languages other than Python**. It is necessary to **re-use** these libraries since:

- **Resources** are not available to re-implement work that is sometimes the result of multiple decades of effort from multiple researchers.
- The **scientific endeavor** is built on the practice of **reproducing** and **building on the top** of the efforts of our predecessors.

The *lingua franca* of computing is the **C programming language** because most operating systems themselves are written in C.

As a consequence,

- **Native binaries** reflect characteristics and compatibility with of the C language
- The reference implementation of Python, *CPython*, is implemented in C
- CPython supports **binary extension modules written in C**
- Most other pre-compiled programming languages have a **compatibility layer with C**
- CPython is an excellent language to **integrate scientific codes**!

Common programming languages compiled into native libraries for scientific computing include:

- Fortran
- C
- C++
- Cython
- Rust

Build Components and Requirements

Build component categories:

build tools Tools use in the build process, such as the compiler, linker, build systems, system introspection tool, and package manager

Example compilers:

- GCC
- Clang
- Visual Studio

Compilers translate source code from a human readable to a machine readable form.

Example linkers:

- ld
- ld.gold
- link.exe

Linkers combine the results of compilers into a shared library that is executed at program runtime.

Example build systems:

- distutils.build_ext
- Unix Makefiles
- Ninja
- MSBuild in Visual Studio

Builds systems coordinate invocation of the compiler and linker, passing flags, and only out-of-date build targets are built.

Example system introspection tools:

- CMake
- GNU Autotools
- Meson

System introspection tools examine the host system for available build tools, the location of build dependencies, and properties of the build target to generate the appropriate build system configuration files.

Example package managers:

- conda
- pip
- apt
- yum
- chocolatey
- homebrew

Package managers resolve dependencies so the required build host artifacts are available for the build.

build host artifacts These are files required on the *host* system performing the build. This includes **header files**, **.h* files, which define the C program **symbols**, i.e. variable and function names, for the native binary with which we want to integrate. This also usually includes the native binaries themselves, i.e. the **executable or shared library**. An important exception to this rule is *libpython*, which we do not need on some platforms due to weak linking rules.

target system artifacts These are artifacts intended to be run on the **target** system, typically the shared library C-extension.

When the build *host* system is different from the *target* system, we are **cross-compiling**.

For example, when we are building a Linux Python package on macOS is cross-compiling. In this case macOS is the *host* system and Linux is the *target* system.

Distributable binaries must use a **compatible C-runtime**.

The table below lists the different C runtime implementations, compilers and their usual distribution mechanisms for each operating systems.

	Linux	MacOSX	Windows
C runtime	GNU C Library (glibc)	libSystem library	Microsoft C run-time library
Compiler	GNU compiler (gcc)	clang	Microsoft C/C++ Compiler (cl.exe)
Provenance	Package manager	OSX SDK within XCode	• Microsoft Visual Studio • Microsoft Windows SDK

Linux C-runtime compatibility is determined by the version of **glibc** used for the build.

The glibc library shared by the system is forwards compatible but not backwards compatible. That is, a package built on an older system *will* work on a newer system, while a package built on a newer system will not work on an older system.

The manylinux project provides Docker images that have an older version of glibc to use for distributable Linux packages.

The C-runtime on macOS is determined by a build time option, the *osx deployment target*, which defines the minmum version of macOS to support, e.g. *10.9*.

A macOS system comes with support for running building binaries for its version of OSX and older versions of OSX.

The XCode toolchain comes with SDK's that support multiple target versions of OSX.

When building a wheel, this can be specified with *–plat-name*:

```
python setup.py bdist_wheel --plat-name macosx-10.6-x86_64
```

The C-runtime used on Windows is associated with the version of Visual Studio.

	Architecture	
CPython Version	x86 (32-bit)	x64 (64-bit)
3.5 and above	Visual Studio 14 2015	Visual Studio 14 2015 Win64
3.3 to 3.4	Visual Studio 10 2010	Visual Studio 10 2010 Win64
2.7 to 3.2	Visual Studio 9 2008	Visual Studio 9 2008 Win64

Distributable binaries are also built to be compatible with a certain CPU architecture class. For example

- x86_64 (currently the most common)
- x86
- ppc64le

Scientific Python Build Tools

scikit-build is an improved build system generator for CPython C/C++/Fortran/Cython extensions.

scikit-build provides better support for additional compilers, build systems, cross compilation, and locating dependencies and their associated build requirements.

The **scikit-build** package is fundamentally just glue between the *setuptools* Python module and CMake.

To build and install a project configured with scikit-build:

```
pip install .
```

To build and install a project configured with scikit-build for development:

```
pip install -e .
```

To build and package a project configured with scikit-build:

```
pip wheel -w dist .
```

Conda is an open source package management system and environment management system that runs on Windows, macOS and Linux.

Conda quickly installs, runs and updates packages and their dependencies. Conda easily creates, saves, loads and switches between environments on your local computer.

Conda was created for Python programs, but it can package and distribute software for any language.

scikit-build and *conda* **abstract away** and **manage platform-specific details** for you!

Exercises

Exercise 1: Build a Python Package with a C++ Extension Module

Download the hello-cpp example C++ project and build a wheel package with the commands:

```
cd hello-cpp
pip wheel -w dist --verbose .
```

Examine files referenced in the build output. What is the purpose of all referenced files?

Exercise 2: Build a Python Package with a Cython Extension Module

Download the hello-cython example C++ project and build a wheel package with the commands:

```
cd hello-cython
pip wheel -w dist --verbose .
```

Examine files referenced in the build output. What is the purpose of all referenced files?

Bonus Exercise 3: Build a Distributable Linux Wheel Package

If Docker is installed, create a dockcross manylinux bash driver script. From a bash shell, run:

```
# cd into the hello-cpp project from Exercise 1
cd hello-cpp
docker run --rm dockcross/manylinux-x64 > ./dockcross-manylinux-x64
chmod +x ./dockcross-manylinux-x64
```

The *dockcross* driver script simplifies execution of commands in the isolated Docker build environment that use sources in the current working directory.

To build a distributable Python 3.6 Python wheel, run:

```
./dockcross-manylinux-x64 /opt/python/cp36-cp36m/bin/pip wheel -w dist .
```

Which will output:

```
Processing /work
Building wheels for collected packages: hello-cpp
Running setup.py bdist_wheel for hello-cpp ... done
Stored in directory: /work/dist
Successfully built hello-cpp
```

and produce the wheel:

```
./dist/hello_cpp-1.2.3-cp36-cp36m-linux_x86_64.whl
```

To find the version of glibc required by the extension, run:

```
./dockcross-manylinux-x64 bash -c 'cd dist && unzip -o hello_cpp-1.2.3-cp36-cp36m-
↪linux_x86_64.whl && objdump -T hello/_hello.cpython-36m-x86_64-linux-gnu.so | grep␣
↪GLIBC'
```

What glibc version compatibility is required for this binary?

manylinux: https://github.com/pypa/manylinux

Bonus Exercise 4: Setting up continuous integration

- See branch master-with-ci branch associated with `hello-cpp` example:
  - Use scikit-ci for simpler and centralized CI configuration for Python extensions.
  - Use scikit-ci-addons, a set of scripts useful to help drive CI.
  - On CircleCI, use manylinux dockcross images including scikit-build, cmake and ninja packages.

1.1.6 Conda Packages

Building Conda Packages

A package system for anything…

Wheels vs. Conda packages

Wheels	Conda packages
Employed by pip, blessed by PyPA	Foundation of Anaconda ecosystem
Used by any python installation	Used by conda python installations
Mostly specific to Python ecosystem	General purpose (any ecosystem)
Good mechanism for specifying range of python compatibility	Primitive support for multiple python versions (noarch)
Depends on static linking or other system package managers to provide core libraries	Can bundle core system-level shared libraries as packages, and resolve dependencies

Introducing conda-build

- Orchestrates environment creation, activation, and build/test processes
- Can build conda packages and/or wheels
- Separate project from conda, but very tightly integrated
- Open-source, actively developed:

  https://github.com/conda/conda-build

Excercise: let's use `conda-build`

```
conda install conda-build
```

- Windows only:

```
conda install m2-patch posix
```

- All platforms:

```
cd python-packaging-tutorial/conda_build_recipes
conda build 01_minimum
```

What happened?

- Templates filled in, recipe interpreted
- Build environment created (isolated)
- Build script run
- New files in build environment bundled into package
- Test environment created (isolated)
- Tests run on new package
- cleanup

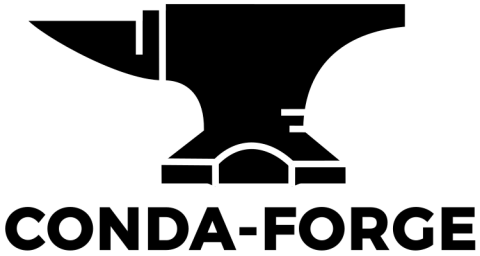
Obtaining recipes

- Existing recipes (best)
  - https://github.com/AnacondaRecipes
  - https://github.com/conda-forge
- Skeletons from other repositories (PyPI, CRAN, CPAN, RPM)
- DIY

Anaconda Recipes

- https://github.com/AnacondaRecipes
- Official recipes that Anaconda uses for building packages
- Since Anaconda 5.0, forked from conda-forge recipes.
- Intended to be compatible with conda-forge long-term
- Presently, ahead of conda-forge on use of conda-build 3 features

Conda-forge



https://conda-forge.org

- https://conda-forge.org
- Numfocus-affiliated community organization made up of volunteers
- One github repository per recipe
  - Fine granularity over permissions
- Heavy use of automation for building, deploying, and updating recipes
- Free builds on public CI services (TravisCI, CircleCI, Appveyor)

Skeletons

- Read metadata from upstream repository
- Translate that into a recipe

- **Will** save you some boilerplate work
- **Might** work out of the box
  - (should not assume automatic, though)

conda skeleton

conda skeleton pypi:

```
conda skeleton pypi <package name on pypi>

conda skeleton pypi click

conda skeleton pypi --recursive pyinstrument
```

conda skeleton cran

```
conda skeleton cran <name of pkg on cran>

conda skeleton cran acs

conda skeleton cran --recursive biwt
```

When all else fails, write a recipe

Only required section:

```
package:
  name: abc
  version: 1.2.3
```

Exercise: create a basic recipe

https://github.com/python-packaging-tutorial/python-packaging-tutorial/tree/master/conda_build_recipes/01_minimum

Source types

- url
- git
- hg
- svn
- local path

meta.yaml source section

Exercise: point your recipe at local files

https://github.com/python-packaging-tutorial/python-packaging-tutorial/tree/master/conda_build_recipes/02_local_source

Building packages

Lots of ways, but let's start simple:

- build.sh (unix)
- bld.bat (windows)

Filenames are of paramount importance here

build.sh: stuff to run on mac/linux

- It's a shell script: do what you want
- Snapshot files in $PREFIX before running script; again after
- Files that are new in $PREFIX are what make up your package
- Several useful env vars for use in build.sh: https://conda.io/docs/user-guide/tasks/build-packages/environment-variables.html

bld.bat: stuff to run on windows

- It's a batch script: do what you want
- Snapshot files in %PREFIX% before running script; again after
- Files that are new in %PREFIX% are what make up your package
- Several useful env vars for use in bld.bat: https://conda.io/docs/user-guide/tasks/build-packages/environment-variables.html

Exercise: Copy a file into the package

https://github.com/python-packaging-tutorial/python-packaging-tutorial/tree/master/conda_build_recipes/03_copy_file

Build options

`number:` version reference of recipe (as opposed to version of source code)

`script:` quick build steps, avoid separate build.sh/bld.bat files

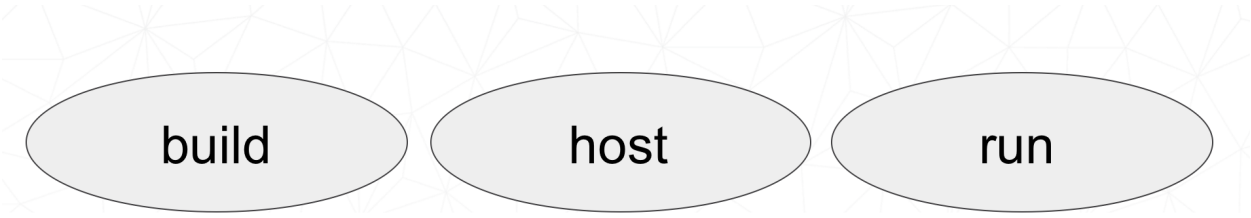
`skip:` skip building recipe on some platforms

`entry_points:` python code locations to create executables for

`run_exports:` add dependencies to downstream consumers to ensure compatibility

meta.yaml build section

Requirements



Build requirements

- Tools to build packages with; things that don't directly go into headers or linking
- Compilers
- autotools, pkg-config, m4, cmake
- archive tools

Host requirements

- External dependencies for the package that need to be present at build time
- Headers, libraries, python/R/perl
- Python deps used in setup.py
- Not available at runtime, unless also specified in run section

Run requirements

- Things that need to be present when the package is installed on the end-user system
- Runtime libraries
- Python dependencies at runtime
- Not available at build time unless also specified in build/host section

Requirements: build vs. host

- Historically, only build
- Still fine to use only build
- host introduced for cross compiling
- host also useful for separating build tools from packaging environment

If in doubt, put everything in host

- build is treated same as host for old-style recipes (only build, no `{{ compiler() }}`)
- packages are bundled from host env, not build env

Exercise: use Python in a build script

https://github.com/python-packaging-tutorial/python-packaging-tutorial/tree/master/conda_build_recipes/04_python_in_build

Post-build Tests

- Help ensure that you didn't make a packaging mistake
- Ideally checks that necessary shared libraries are included as dependencies

Dependencies

Describe dependencies that are required for the tests (but not for normal package usage)

```
test:
  requires:
    - pytest
```

Post-build tests: test files

All platforms: `run_test.pl`, `run_test.py`, `run_test.r`, `run_test.lua`

Windows: `run_test.bat`

Linux / Mac: `run_test.sh`

Post-build tests

- May have specific requirements
- May specify files that must be bundled for tests (`source_files`)
- `imports`: language specific imports to try, to verify correct installation
- `commands`: sequential shell-based commands to run (not OS-specific)

https://conda.io/docs/user-guide/tasks/build-packages/define-metadata.html#test-section

Import Tests

```
test:
  imports:
        - dateutil
        - dateutil.rrule
```

(continues on next page)

(continued from previous page)

```
        - dateutil.parser
        - dateutil.tz
```

Test commands

```
test:
  commands:
      - curl --version
      - curl-config --features  # [not win]
      - curl-config --protocols  # [not win]
      - curl https://some.website.com
```

Exercise: add some tests

https://github.com/python-packaging-tutorial/python-packaging-tutorial/tree/master/conda_build_recipes/05_test_python

Outputs - more than one pkg per recipe

```
package:
  name: some-split
  version: 1.0

outputs:
  - name: subpkg
  - name: subpkg2
```

- Useful for consolidating related recipes that share (large) source
- Reduce update burden
- Reduce build time by keeping some parts of the build, while looping over other parts
- Also output different types of packages from one recipe (wheels)

https://conda.io/docs/user-guide/tasks/build-packages/define-metadata.html#outputs-section

About section

Extra section: free-for-all

- Used for external tools or state management
- No schema
- Conda-forge's maintainer list
- Conda-build's notion of whether a recipe is "final"

https://conda.io/docs/user-guide/tasks/build-packages/define-metadata.html#extra-section

Conditional lines (selectors)

```
some_content     # [some expression]
```

- content inside `[...]` is eval'ed
- namespace includes OS info, python info, and a few others

https://conda.io/docs/user-guide/tasks/build-packages/define-metadata.html#preprocessing-selectors

Exercise: Limit a Recipe to Only Linux

```
package:
  name: example_skip_recipe
  version: 1.0

  build:
  skip: True
```

```
package:
  name: example_skip_recipe
  version: 1.0

  build:
  skip: True  # [not linux]
```

Intro to Templating with Jinja2

- Fill in information dynamically
  - git tag info
  - setup.py recipe data
  - centralized version numbering
  - string manipulation

How does Templating Save You Time?

```
{% set version = "3.0.2" %}

package:
  name: example
  version: {{ version }}
source:
  url: https://site/{{version}}.tgz
```

Jinja2 Templating in `meta.yaml`

Set variables:

```
{% set somevar="someval" %}
```

Use variables:

```
{{ somevar }}
```

Expressions in `{{ }}` are roughly python

Jinja2 conditionals

Selectors are one line only. When you want to toggle a block, use jinja2:

```
{%- if foo -%}

toggled content

on many lines

{% endif %}
```

Exercise: use Jinja2 to reduce edits

```
package:
  name: abc
  version: 1.2.3
```

(continues on next page)

(continued from previous page)

```
source:
  url: http://my.web/abc-1.2.3.tgz
```

```
{% set version="1.2.3" %}
package:
  name: abc
  version: {{ version }}

source:
  url: http://w/abc-{{version}}.tgz
```

Variants: Jinja2 on steroids

Matrix specification in yaml files

```
somevar:
  - 1.0
  - 2.0

anothervar:
  - 1.0
```

All variant variables exposed in jinja2

In meta.yaml,

`{{ somevar }}`

And this loops over values

Exercise: try looping

meta.yaml:

```
package:
  name: abc
  version: 1.2.3

build:
  skip: True # [skipvar]
```

conda_build_config.yaml:

```
skipvar:
  - True
  - False
```

meta.yaml:

```
package:
  name: abc
  version: 1.2.3

requirements:
  build:
    - python {{ python }}

  run:
    - python {{ python }}
```

conda_build_config.yaml:

```
python:
  - 2.7
  - 3.6
```

meta.yaml:

```
package:
  name: abc
  version: 1.2.3

requirements:
  build:
    - python
  run:
    - python
```

conda_build_config.yaml:

```
python:
  - 2.7
  - 3.6
```

Jinja2 functions

loading source data:

`load_setup_py_data`

`load_file_regex`

Dynamic Pinning:

`pin_compatible`

`pin_subpackage`

Compatibility Control:

`compiler`

`cdt`

Loading setup.py data

```
{% set setup_data = load_setup_py_data() %}

package:
  name: abc
  version: {{ setup_data['version'] }}
```

- Primarily a development recipe tool - release recipes specify version instead, and template source download link
- Centralizing version info is very nice - see also `versioneer`, `setuptools_scm`, `autover`, and many other auto-version tools

Loading arbitrary data

```
{% set data = load_file_regex(load_file='meta.yaml',
               regex_pattern='git_tag: ([\\d.]+)') %}

package:
  name: conda-build-test-get-regex-data
  version: {{ data.group(1) }}
```

- Useful when software provides version in some arbitrary file
- Primarily a development recipe tool - release recipes specify version instead, and template source download link

Dynamic pinning

Use in meta.yaml, generally in requirements section:

```
requirements:
  host:
    - numpy
  run:
    - {{ pin_compatible('numpy') }}
```

Use in meta.yaml, generally in requirements section:

```
requirements:
  host:
    - numpy
  run:
    - {{ pin_compatible('numpy') }}
```

- Pin run req based on what is present at build time

Dynamic pinning in practice

Used a lot with numpy:

https://github.com/AnacondaRecipes/scikit-image-feedstock/blob/master/recipe/meta.yaml

Dynamic pinning within recipes

Refer to other outputs within the same recipe

- When intradependencies exist
- When shared libraries are consumed by other libraries

https://github.com/AnacondaRecipes/aggregate/blob/master/clang/meta.yaml

Compilers

Use in meta.yaml in requirements section:

```
requirements:
    build:
        - {{ compiler('c') }}
```

- explicitly declare language needs
- compiler packages can be actual compilers, or just activation scripts
- Compiler packages utilize run_exports to add necessary runtime dependencies automatically

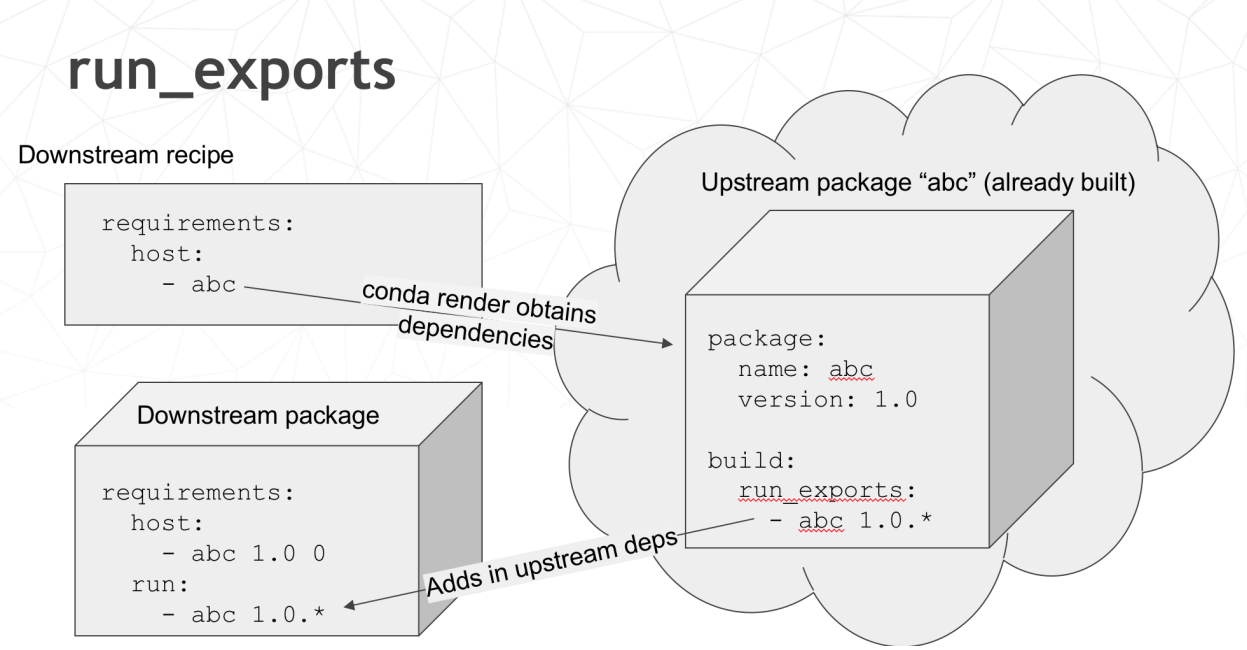
Why put compilers into Conda?

- Explicitly declaring language needs makes reproducing packages with recipe simpler
- Binary compatibility can be versioned and tracked better
- No longer care what the host OS used to build packages is
- Can still use system compilers - just need to give conda-build information on metadata about them. Opportunity for version check enforcement.

`run_exports`

"if you build and link against library abc, you need a runtime dependency on library abc"

This is annoying to keep track of in recipes.



- Add host or run dependencies for downstream packages that depend on upstream that specifies run_exports
- Expresses idea that “if you build and link against library abc, you need a runtime dependency on library abc”
- Simplifies version tracking

Exercise: make a run_exports package

https://github.com/python-packaging-tutorial/python-packaging-tutorial/tree/master/conda_build_recipes/06_has_run_exports

Exercise: use a run_exports package

https://github.com/python-packaging-tutorial/python-packaging-tutorial/tree/master/conda_build_recipes/07_uses_run_exports

Uploading packages: anaconda.org

- Sign-up:
  - `https://anaconda.org/`
- Requirement:
  - `conda install anaconda-client`
- CLI: anaconda upload path-to-package
- conda-build auto-upload:
  - `conda config --set anaconda_upload True`

Fin

Extra slides

Source Patches

- patch files live alongside meta.yaml
- create patches with:
  - `diff`
  - `git diff`
  - `git format-patch`

meta.yaml source section

Exercise: let's make a patch

```
package:
  name: test-patch
  version: 1.2.3

source:
  url: https://zlib.net/zlib-1.2.11.tar.gz

build:
  script: exit 1
```

- Builds that fail leave their build folders in place
- look in output for source tree in:

  `*/conda-bld/test-patch_<numbers>/work`
- `cd` there

```
git init

git add *

git commit -am "init"

edit file of choice

git commit -m "changing file because ..."

git format-patch HEAD~1
```

- copy that patch back alongside meta.yaml
- modify meta.yaml to include the patch

Multiple sources

```
source:
  - url: https://package1.com/a.tar.bz2
    folder: stuff
  - url: https://package1.com/b.tar.bz2
    folder: stuff
    patches:
      - something.patch
  - git_url: https://github.com/conda/conda-build
    folder: conda-build
```

meta.yaml source section

Outputs rules

- List of dicts
- Each list must have `name` or `type` key
- May use all entries from `build`, `requirements`, `test`, `about` sections
- May specify files to bundle either using globs or by running a script

Outputs Examples

https://github.com/AnacondaRecipes/curl-feedstock/blob/master/recipe/meta.yaml

https://github.com/AnacondaRecipes/aggregate/blob/master/ctng-compilers-activation-feedstock/recipe/meta.yaml

Exercise: Split a Package

Curl is a library and an executable. Splitting them lets us clarify where Curl is only a build time dependency, and where it also needs to be a runtime dependency.

Starting point:

https://github.com/conda-forge/curl-feedstock/tree/master/recipe

Solution:

https://github.com/AnacondaRecipes/curl-feedstock/tree/master/recipe

If we have time: conda-forge

1.1.7 Tutorial Content Updates

You will find here the list of changes integrated in the tutorial after it was first given at the SciPy 2018 conference.

Changes are grouped in sections identified using `YYYY-MM` representing the year and month when the related changes were done.

The sections are ordered from most recent to the oldest.

2018-08

Better handling data file in Exercise: A Small Example Package section

- Put package data in *data* directory.
- Reflect this change in the code.
- Add *package_data* to setup function.

2018-07

This is the first set of changes incorporating the feedback from attendees.

Making a Python Package

- Add directory setup_example/capitalize discussed in *Exercise: A Small Example Package* section.

Building and Uploading to PyPI

- Update *Installing a wheel* tutorial adding *Install a package from TestPyPI* section.

1.2 Your Guides

Michael Sarahan: Conda-build tech lead, Anaconda, Inc.

Matt McCormick: Maintainer of dockcross, of Python packages for the Insight Toolkit (ITK)

Jean-Christophe Fillion-Robin: Maintainer of scikit-build, scikit-ci, scikit-ci-addons and python-cmake-buildsystem

Filipe Fernandes: Conda-forge core team, Maintainer of folium and a variety of libraries for ocean sciences.

Chris Barker: Python instructor for the Univ. Washington Continuing Education Program, Contributor to conda-forge project. Lead developer for assorted oceanography / oil spill packages.

Jonathan Helmus: Conda-forge core team. Maintainer of Berryconda. Anaconda, Inc. Builds Tensorflow for fun.