conda-build Documentation

Release 3.19.3+105.gdc6143f5.dirty

Anaconda, Inc.

Aug 05, 2020

CONTENTS

1 Installing and updating conda-build 3

2 Concepts 5

3 User guide 15

4 Resources 47

5 Release notes 109

Index 121

Conda-build contains commands and tools to use conda to build your own packages. It also provides helpful tools to constrain or pin versions in recipes. Building a conda package requires *installing conda-build* and creating a conda *recipe*. You then use the `conda build` command to build the conda package from the conda recipe.

You can build conda packages from a variety of source code projects, most notably Python. For help packing a Python project, see the Setuptools documentation.

OPTIONAL: If you are planning to upload your packages to Anaconda Cloud, you will need an Anaconda Cloud account and client.

CHAPTER

ONE

INSTALLING AND UPDATING CONDA-BUILD

To enable building conda packages:

- install conda
- install conda-build
- update conda and conda-build

1.1 Installing conda-build

To install conda-build, in your terminal window or an Anaconda Prompt, run:

```
conda install conda-build
```

1.2 Updating conda and conda-build

Keep your versions of conda and conda-build up to date to take advantage of bug fixes and new features.

To update conda and conda-build, in your terminal window or an Anaconda Prompt, run:

```
conda update conda
conda update conda-build
```

For release notes, see the conda-build GitHub page.

CHAPTER

TWO

CONCEPTS

2.1 Conda channels

The `conda-build` options `-c CHANNEL` or `--channel CHANNEL` configure additional channels to search for packages.

These are URLs searched in the order they are given (including file:// for local directories).

Then, the defaults or channels from `.condarc` are searched (unless `--override-channels` is given).

You can use 'defaults' to get the default packages for conda, and 'system' to get the system packages, which also takes `.condarc` into account.

You can also use any name and the `.condarc channel_alias` value will be prepended. The default `channel_alias` is http://conda.anaconda.org/.

The option `--override-channels` tells to not search default or `.condarc` channels. Requires the `--channel` or `-c` option.

2.1.1 Identical channel and package name problem

If the channel and package name are identical, it's possible to encounter a build problem if the short channel name is used.

Let's say your Anaconda.org username or an organization name is `example`. And suppose you created a package `example`, whose files' layout is similar to:

```
setup.py
conda/meta.yaml
example/
```

If your build depends on some other packages inside your channel, you will need to add `-c example`, however, the following code:

```
conda-build ./conda/ -c example
```

will fail with the following error message (the path will be different):

```
requests.exceptions.HTTPError: 404 Client Error: None for url:
file:///path/to/your/local/example/noarch/repodata.json
[...]
The remote server could not find the noarch directory for the requested channel with
url: file:///path/to/your/local/example/noarch/repodata.json
[...]
```

(continues on next page)

(continued from previous page)

```
As of conda 4.3, a valid channel must contain a `noarch/repodata.json` and
associated `noarch/repodata.json.bz2` file, even if `noarch/repodata.json`
is empty. please request that the channel administrator create
`noarch/repodata.json` and associated `noarch/repodata.json.bz2` files.
```

This happens because `conda-build` will consider the directory `./example/` in your project as a channel. This is by design due to conda's CI servers, where the build path can be long, complicated, and not predictable prior to build.

There are several ways to resolve this issue.

1. Use the url of the desired channel:

   ```
   conda-build ./conda/ -c https://conda.anaconda.org/example/
   ```

2. Run the build from inside the conda recipe directory:

   ```
   cd conda
   conda-build . -c example
   ```

3. Use the label specification workaround:

   ```
   conda-build ./conda/ -c example/label/main
   ```

   which technically is the same as `-c example`, since `main` is the default label, but now it won't by mistake find a channel `example/label/main` on the local filesystem.

2.2 Channels and generating an index

2.2.1 Channel layout

```
.
├── channeldata.json
├── linux-32
|   ├── repodata.json
│   └── package-0.0.0.tar.bz2
├── linux-64
|   ├── repodata.json
│   └── package-0.0.0.tar.bz2
├── win-64
|   ├── repodata.json
│   └── package-0.0.0.tar.bz2
├── win-32
|   ├── repodata.json
│   └── package-0.0.0.tar.bz2
├── osx-64
|   ├── repodata.json
│   └── package-0.0.0.tar.bz2
...
```

2.2.2 Parts of a channel

- Channeldata.json contains metadata about the channel, including:
  - What subdirs the channel contains.
  - What packages exist in the channel and what subdirs they are in.
- Subdirs are associated with platforms. For example, the linux-64 subdir contains packages for linux-64 systems.
- Repodata.json contains an index of the packages in a subdir. Each subdir will have it's own repodata.
- Channels have packages as tarballs under corresponding subdirs.

2.2.3 channeldata.json

```
{
  "channeldata_version": 1,
  "packages": {
    "super-fun-package": {
      "activate.d": false,
      "binary_prefix": false,
      "deactivate.d": false,
      "home": "https://github.com/Home/super-fun-package",
      "license": "BSD",
      "post_link": false,
      "pre_link": false,
      "pre_unlink": false,
      "reference_package": "win-64/super-fun-package-0.1.0-py37_0.tar.bz2",
      "run_exports": {},
      "subdirs": [
        "win-64"
      ],
      "summary": "A fun package! Open me up for rainbows",
      "text_prefix": false,
      "version": "0.1.0"
    },
    "subdirs": [
      "win-64",
      ...
    ]
}
```

2.2.4 repodata.json

```
{
  "packages": {
    "super-fun-package-0.1.0-py37_0.tar.bz2": {
      "build": "py37_0",
      "build_number": 0,
      "depends": [
        "some-depends"
      ],
      "license": "BSD",
      "md5": "a75683f8d9f5b58c19a8ec5d0b7f796e",
      "name": "super-fun-package",
```

(continues on next page)

(continued from previous page)

```
      "sha256": "1fe3c3f4250e51886838e8e0287e39029d601b9f493ea05c37a2630a9fe5810f",
      "size": 3832,
      "subdir": "win-64",
      "timestamp": 1530731681870,
      "version": "0.1.0"
    },
    ...
  }
```

2.2.5 How an index is generated

For each subdir:

- Look at all the packages that exist in the subdir.
- Generate a list of packages to add/update/remove.
- Remove all packages that need to be removed.

For all packages that need to be added/updated:

- Extract the package to access metadata including full package name, mtime, size, and index.json.
- Add package to repodata.

2.2.6 Example: Building a channel

To build a local channel and put a package in it, follow the directions below.

1. Make the channel structure.

   ```
   $ mkdir local-channel
   $ cd local-channel
   $ mkdir linux-64 osx-64
   ```

2. Put your favorite package in the channel.

   ```
   $ wget https://anaconda.org/anaconda/scipy/1.1.0/download/linux-64/scipy-
   ↪1.1.0-py37hfa4b5c9_1.tar.bz2 -P linux-64
   $ wget https://anaconda.org/anaconda/scipy/1.1.0/download/osx-64/scipy-1.
   ↪1.0-py37hf5b7bf4_0.tar.bz2 -P osx-64
   ```

3. Run a conda index. This will generate both channeldata.json for the channel and repodata.json for the linux-64 and osx-64 subdirs, along with some other files.

   ```
   $ conda index .
   ```

4. Check your work by searching the channel.

   ```
   $ conda search -c file:/<path to>/local-channel scipy | grep local-channel
   ```

2.3 Conda-build recipes

- *Conda-build process*
- *Deep dive*
  - *Templates*
  - *Environments*
  - *Building*
  - *Prefix replacement*
  - *Testing*
  - *Output metadata*
- *More information*

To enable building conda packages, *install and update conda and conda-build*.

Building a conda package requires a recipe. A conda-build recipe is a flat directory that contains the following files:

- `meta.yaml`---A file that contains all the metadata in the recipe. Only `package/name` and `package/version` are required.
- `build.sh`---The script that installs the files for the package on macOS and Linux. It is executed using the `bash` command.
- `bld.bat`---The build script that installs the files for the package on Windows. It is executed using `cmd`.
- `run_test.[py,pl,sh,bat]`---An optional Python test file, a test script that runs automatically if it is part of the recipe.
- Optional patches that are applied to the source.
- Other resources that are not included in the source and cannot be generated by the build scripts. Examples are icon files, readme files and build notes.

Tip: When you use the *conda skeleton* command, the first 3 files---`meta.yaml`, `build.sh`, and `bld.bat`---are automatically generated for you.

2.3.1 Conda-build process

Conda-build performs the following steps:

1. Reads the metadata.
2. Downloads the source into a cache.
3. Extracts the source into the source directory.
4. Applies any patches.
5. Re-evaluates the metadata, if source is necessary to fill any metadata values.
6. Creates a build environment and then installs the build dependencies there.

7. Runs the build script. The current working directory is the source directory with environment variables set. The build script installs into the build environment.
8. Performs some necessary post-processing steps, such as shebang and rpath.
9. Creates a conda package containing all the files in the build environment that are new from step 5, along with the necessary conda package metadata.
10. Tests the new conda package if the recipe includes tests:
    1. Deletes the build environment and source directory to ensure that the new conda package does not inadvertantly depend on artifacts not included in the package.
    2. Creates a test environment with the package and its dependencies.
    3. Runs the test scripts.

The conda-recipes repo contains example recipes for many conda packages.

Caution: All recipe files, including `meta.yaml` and build scripts, are included in the final package archive that is distributed to users. Be careful not to put sensitive information such as passwords into recipes where it could be made public.

The `conda skeleton` command can help to make skeleton recipes for common repositories, such as PyPI.

2.3.2 Deep dive

Let's take a closer look at how conda-build uses a recipe to create a package.

Templates

When you build a conda package, conda-build renders the package by reading a template in the meta.yaml. See *Templating with Jinja*.

Templates are filled in using your conda-build config, which shows the matrix of things to build against. The `conda build config` determines how many builds it has to do. For example, defining a conda_build_config.yaml of the form and filling it defines a matrix of 4 packages to build:

```
foo:
  - 1.0
  - 2.0
bar:
  - 1.2.0
  - 1.4.0
```

After this, conda-build determines what the outputs will be. For example, if your `conda build config` indicates that you want 2 different versions of Python, conda-build will show you the rendering for each Python version.

Environments

To build the package, conda-build will make an environment for you and install all of the build and run dependencies in that environment. Conda-build will indicate where you can successfully build the package. The prefix will take the form:

```
<path to conda>/conda-bld/<package name and string>/h_env_placeholder...
```

Conda-forge downloads your package source and then builds the conda package in the context of the build environment. For example, you may direct it to download from a Git repo or pull down a tarball from another source. See the *Source section* for more information.

What conda-build puts into a package depends on what you put into the build, host, or run sections. See the *Requirements section* for more information. Conda-build will use this information to identify dependencies to link to and identify the run requirements for the package. This allows conda-build to understand what is needed to install the package.

Building

Once the content is downloaded, conda-build runs the build step. See the *Build section* for more information. The build step runs a script. It can be one that you provided. See the *Script* section for more information.

If you do not define the script section, then you can create a build.sh or a bld.bat file to be run.

Prefix replacement

When the build environment is created, it is in a placeholder prefix. When the package is all bundled up, the prefix is set to a dummy prefix. When conda is ready to install the package, it rewrites the dummy prefix with the correct one.

Testing

Once a package is built, conda-build will test it. To do this, it creates another environment and installs the conda package. The form of this prefix is:

```
<path to conda>/conda-bld/<package name + string>/_test_env_placeholder...
```

At this point, conda-build has all of the info from the meta.yaml about what its runtime dependencies are, so those dependencies are installed as well. This generates a test runner script with a reference to the testing meta.yaml that is created. See the *Test section* for more information. That file is run for testing.

Output metadata

After the package is built and tested, conda-build cleans up the environments created prior and outputs the metadata. The recipe for the package is also added in the output metadata. The metadata directory is on the top level of the tarball in the `info` directory. The metadata contains information about the dependencies of the package and a list of where all of the files in the package go when it is installed. Conda reads that metadata when it needs to install.

Running `conda install` causes conda to:

- reach out to the repo data containing the dependencies,
- guess the right dependencies,
- install a list of packages,

- unpack the tarball to look at the info,
- verify the file based on metadata in the package, and then
- go through each file in the package and put it in the right location.

2.3.3 More information

Review *Defining metadata (meta.yaml)* to see a breakdown of the components of a recipe, including:

- Package name.
- Package version.
- Descriptive metadata.
- Where to obtain source code.
- How to test the package.

2.4 Package naming conventions

To facilitate communication and documentation, conda observes the package naming conventions listed below.

2.4.1 Package name

The name of a package, without any reference to a particular version. Conda package names are normalized and they may contain only lowercase alpha characters, numeric digits, underscores, hyphens, or dots. In usage documentation, these are referred to by `package_name`.

2.4.2 Package version

A version number or string, often similar to `X.Y` or `X.Y.Z`, but it may take other forms as well.

2.4.3 Build string

An arbitrary string that identifies a particular build of a package for conda. It may contain suggestive mnemonics, but these are subject to change, and you should not rely on it or try to parse it for any specific information.

2.4.4 Canonical name

The package name, version, and build string joined together by hyphens---name-version-buildstring. In usage documentation, these are referred to by `canonical_name`.

2.4.5 Filename

Conda package filenames are canonical names, plus the suffix `.tar.bz2` or `.conda`.

The following figure compares a canonical name to a filename:

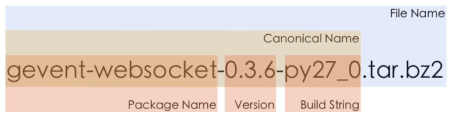

Fig. 1: Conda package naming

Conda supports both `.conda` and `.tar.bz2` package extensions. The `.conda` format is generally smaller and more efficient than `.tar.bz2` packages. Read our blog post about it to learn more.

The build string is created as the package is built. Things that contribute to it are the variants specified either by the command line or the configuration from the `conda_build_config.yaml`, and the build number in the recipe. If there are no variants, then the build string is the build number that is specified in the recipe.

2.4.6 Package specification

A package name together with a package version---which may be partial or absent---joined by an equal sign.

EXAMPLES:

- `python=2.7.3`
- `python=2.7`
- `python`

In usage documentation, these are referred to by `package_spec`.

2.5 What is a "package"?

- A package is anything you install using your package manager.
- A "conda package" is a compressed tarball that contains
  - the module to be installed
  - information on how to install the package
- You can use conda-build to build a conda package.

2.6 What about channels

- Channels contain packages.
- They conform to a standard structure and contain an index of available packages.
- An index of the available packages can be generated by running:

```
$ conda index <path to channel>
```

- conda is able to install from channels and uses the indexes in the channel to solve for requirements and dependencies.

2.7 Building Anaconda installers

- Anaconda(/Miniconda) installers are built with a modified version of constructor.
- The idea is to build an Anaconda metapackage and bundle it together with some other packages to build an Anaconda installer.

CHAPTER

THREE

USER GUIDE

Welcome to the conda-build user guide. Here you can find tutorials and recipes as well as information about environment variables and wheel files.

3.1 Getting started

Before starting the tutorials, consider reviewing *how to install and update conda-build* and *conda-build concepts*.

You may also find our *resources* collection helpful.

3.1.1 Prerequisites

Before starting the tutorials, you will need to install Miniconda or Anaconda, conda-build, and Git.

After you've installed Miniconda or Anaconda, you can use conda to install conda-build and Git.

3.1.2 Tutorial submissions

Have an idea for a tutorial? You can submit your suggestions to Anaconda by emailing us at documentation@anaconda.com.

To create your own tutorials, follow the *writing style guide* and *tutorial template*.

3.2 Tutorials

Before starting the tutorials, review the *Getting started* guide.

3.2.1 Building conda packages

- *Overview*
- *Who is this for?*
- *Before you start*
- *Toolkit*

- *Developing a build strategy*
- *Building with a Python version different from your Miniconda installation*
- *Automated testing*
- *Building a SEP package with conda and Python 2 or 3*
- *Building a GDAL package with conda and Python 2 or 3*

Overview

This tutorial describes how to use conda-build to create conda packages on Windows, macOS, and Linux using the examples of SEP and GDAL. Additional Windows-specific instructions are provided in the *Toolkit* section.

The final built packages from this tutorial are available on Anaconda Cloud:

- SEP.
- GDAL.

This tutorial also describes writing recipes. You can see the final SEP recipe and the GDAL recipe on GitHub in the conda-build documentation repository.

Who is this for?

This tutorial is for Windows, macOS, and Linux users who wish to build more complex conda packages. This tutorial will involve building scientific packages, which require compilers for several different Python versions.

Before you start

Before you start, make sure you have installed:

- Conda.
- *Conda-build*.
- Any compilers you want.

Toolkit

Microsoft Visual Studio

In the standard practices of the conda developers, conda packages for different versions of Python are each built with their own version of Visual Studio (VS):

- Python 2.7 packages with Visual Studio 2008
- Python 3.4 packages with VS 2010
- Python 3.5 packages with VS 2015, (default) 2017
- Python 3.6 packages with VS 2015, (default) 2017
- Python 3.7 packages with VS 2015, (default) 2017

Using these versions of VS to build packages for each of these versions of Python is also the practice used for the official python.org builds of Python. Currently VS 2008 and VS 2010 are available only through resellers, while VS 2015 and VS 2017 can be purchased online from Microsoft. Note there is also a community edition of VS 2015 and VS 2017 which may be used.

Alternatives to Microsoft Visual Studio

There are free alternatives available for each version of the VS compilers:

- Instead of VS 2008, it is often possible to substitute the free Microsoft Visual C++ Compiler for Python 2.7.
- Instead of VS 2010, it is often possible to substitute the free Microsoft Windows SDK for Windows 7 and .NET Framework 4.
- Make sure that you also install VS 2010 Service Pack 1 (SP1).
- Due to a bug in the VS 2010 SP1 installer, the compiler tools may be removed during installation of VS 2010 SP1. They can be restored as described in Microsoft Visual C++ 2010 Service Pack 1 Compiler Update for the Windows SDK 7.1.
- Visual Studio 2015 has a full-featured, free Community edition for academic research, open source projects, and certain other use cases.

The MS Visual C++ Compiler for Python 2.7 and the Microsoft Windows SDK for Windows 7 and .NET Framework 4 are both reasonably well tested. Conda-build is carefully tested to support these configurations, but there are known issues with the CMake build tool and these free VS 2008 and 2010 alternatives. In these cases, you should prefer the "NMake Makefile" generator, rather than a Visual Studio solution generator.

Windows versions

You can use any recent version of Windows. These examples were built on Windows 10.

Other tools

Some environments initially lack tools such as patch or Git that may be needed for some build workflows.

On Windows, these can be installed with conda:

```
conda install git m2-patch
```

On macOS and Linux, replace `m2-patch` with patch.

Developing a build strategy

Conda recipes are typically built with a trial-and-error method. Often the first attempt to build a package fails with compiler or linker errors, often caused by missing dependencies. The person writing the recipe then examines these errors and modifies the recipe to include the missing dependencies, usually as part of the `meta.yaml` file. Then the recipe writer attempts the build again and, after a few of these cycles of trial and error, the package builds successfully.

Building with a Python version different from your Miniconda installation

Miniconda2 and Miniconda3 can each build packages for either Python 2 or Python 3 simply by specifying the version you want. Miniconda2 includes only Python 2 and Miniconda3 includes only Python 3.

Installing only one makes it easier to keep track of the builds, but it is possible to have both installed on the same system at the same time. If you have both installed, use the `where` command on Windows, or `which` command on Linux to see which version comes first on PATH since this is the one you will be using:

```
where python
```

To build a package for a Python version other than the one in your Miniconda installation, use the `--python` option in the `conda-build` command.

EXAMPLE: To build a Python 3.5 package with Miniconda2:

```
conda-build recipeDirectory --python=3.5
```

Note: Replace `recipeDirectory` with the name and path of your recipe directory.

Automated testing

After the build, if the recipe directory contains a test file. This test file is named `run_test.bat` on Windows, `run_test.sh` on macOS or Linux, or `run_test.py` on any platform. The file runs to test the package and any errors are reported. After seeing "check the output," you can also test if this package was built by using the command:

```
$ conda build --test <path to package>.tar.bz2
```

Note: Use the *Test section of the meta.yaml file* to move data files from the recipe directory to the test directory when the test is run.

Building a SEP package with conda and Python 2 or 3

The SEP documentation states that SEP runs on Python 2 and 3, and it depends only on NumPy. Searching for SEP and PyPI shows that there is already a PyPI package for SEP.

Because a PyPI package for SEP already exists, the `conda skeleton` command can make a skeleton or outline of a conda recipe based on the PyPI package. Then the recipe outline can be completed manually and conda can build a conda package from the completed recipe.

Install Visual Studio

If you have not already done so, install the appropriate version of Visual Studio:

- For Python 3---Visual Studio 2017:
  1. Choose Custom install.
  2. Under Programming Languages, choose to install Visual C++.
- For Python 2---Visual Studio 2008:

1. Choose Custom install.
2. Choose to install X64 Compilers and Tools. Install Service Pack 1.

Make a conda skeleton recipe

1. Run the skeleton command:

   ```
   conda skeleton pypi sep
   ```

   The `skeleton` command installs into a newly created directory called `sep`.

2. Go to the `sep` directory to view the files:

   ```
   cd sep
   ```

   One skeleton file has been created: `meta.yaml`

Edit the skeleton files

For this package, `bld.bat` and `build.sh` need no changes. You need to edit the `meta.yaml` file to add the dependency on NumPy and add an optional test for the built package by importing it. For more information about what can be specified in `meta.yaml`, see *Defining metadata (meta.yaml)*.

1. In the requirements section of the `meta.yaml` file, add a line that adds NumPy as a requirement to build the package.
2. Add a second line to list NumPy as a requirement to run the package.
3. Set the NumPy version to the letters `x.x`.
4. Make sure the new line is aligned with `- python` on the line above it, so as to ensure proper yaml format.

EXAMPLE:

```
requirements:
  host:
    - python
    - numpy     x.x

  run:
    - python
    - numpy     x.x
```

Notice that there are two types of requirements, host and run. Host represents packages that need to be specific to the target platform when the target platform is not necessarily the same as the native build platform. Run represents the dependencies that should be installed when the package is installed.

Note: Using the letters `x.x` instead of a specific version such as `1.11` pins NumPy dynamically, so that the actual version of NumPy is taken from the build command. Currently, NumPy is the only package that can be pinned dynamically. Pinning is important for SEP because this package uses NumPy's C API through Cython. That API changes between NumPy versions, so it is important to use the same NumPy version at runtime that was used at build time.

OPTIONAL: Add a test for the built package

Adding this optional test will test the package at the end of the build by making sure that the Python statement `import sep` runs successfully:

1. Add `- sep`, checking to be sure that the indentation is consistent with the rest of the file.

   EXAMPLE:

```
test:
  # Python imports
  imports:
    - sep
```

Build the package

Build the package using the recipe you just created:

```
conda build sep
```

Check the output

1. Check the output to make sure that the build completed successfully. The output contains the location of the final package file and a command to upload the package to Anaconda Cloud. The output will look something like:

```
# Automatic uploading is disabled
# If you want to upload package(s) to anaconda.org later, type:
anaconda upload /Users/builder/miniconda3/conda-bld/osx-64/sep-1.0.3-np111py36_0.
↪tar.bz2
# To have conda build upload to anaconda.org automatically, use
# $ conda config --set anaconda_upload yes
anaconda_upload is not set.  Not uploading wheels: []
#################################################################################
↪##
Resource usage summary:
Total time: 0:00:56.4
CPU usage: sys=0:00:00.7, user=0:00:07.0
Maximum memory usage observed: 220.1M
Total disk usage observed (not including envs): 3.9K
#################################################################################
↪##
Source and build intermediates have been left in /Users/builder/miniconda3/conda-
↪bld.
There are currently 437 accumulated.
To remove them, you can run the ```conda build purge``` command
```

2. If there are any linker or compiler errors, modify the recipe and build again.

Building a GDAL package with conda and Python 2 or 3

This procedure describes how to build a package with Python 2 or Python 3. Follow the instructions for your preferred version.

To begin, install Anaconda or Miniconda and conda-build. If you are using a Windows machine, also use conda to install Git and the m2-patch.

```
conda install git
conda install m2-patch
```

Because GDAL includes C and C++, building it on Windows requires Visual Studio. This procedure describes how to build a package with Python 2 or Python 3. Follow the instructions for the version with which you want to build.

To build a GDAL package:

1. Install Visual Studio:
   - For Python 3, install Visual Studio 2017. Choose Custom install. Under Programming Languages, select workloads that come from Visual Studio so you choose the Desktop Development with C++ and Universal Platform C.
   - For Python 2, install Visual Studio 2008. Choose Custom install. Choose to install X64 Compilers and Tools. Install Visual Studio 2008 Service Pack 1.
2. Install Git. Because the GDAL package sources are retrieved from GitHub for the build, you must install Git:
   ```
   conda install git m2-patch conda-build
   ```
3. Get gdal-feedstock. For the purpose of this tutorial, we will be using a recipe from Anaconda:
   ```
   git clone https://github.com/AnacondaRecipes/gdal-feedstock.git
   ```
4. Use conda-build to build the gdal-feedstock:
   ```
   conda build gdal-feedstock
   ```
5. Check the output to make sure the build completed successfully. The output also contains the location of the final package file and a command to upload the package to Cloud. For this package in particular, there should be two packages outputted: libgdal and GDAL.
6. In case of any linker or compiler errors, modify the recipe and run it again.

Let's take a better look at what's happening inside the gdal-feedstock. In particular, what is happening in the `meta.yaml`.

The first interesting bit happens under `source` in the patches section:

```
patches:
  # BUILT_AS_DYNAMIC_LIB.
  - 0001-windowshdf5.patch
  # Use multiple cores on Windows.
  - 0002-multiprocessor.patch
  # disable 12 bit jpeg on Windows as we aren't using internal jpeg
  - 0003-disable_jpeg12.patch
```

This section says that when this package is being built on a Windows platform, apply the following patch files. Notice that the patch files are in the *patches* directory of the recipe. These patches will only be applied to Windows since the `# [win]` selector is applied to each of the patch entries. For more about selectors, see *Preprocessing selectors*.

In the requirements section, notice how there are both a build and host set of requirements. For this recipe, all the compilers required to build the package are listed in the build requirements. Normally, this section will list out packages required to build the package. GDAL requires CMake on Windows, as well as C compilers. Notice that the C compilers are pulled into the recipe using the syntax `{{ compiler('c') }}`. Since conda-build 3, conda-build defines a jinja2 function `compiler()` to specify compiler packages dynamically. So, using the `compiler('c')` function in a conda recipe will pull in the correct compiler for any build platform. For more information about compilers with conda-build see *compiler-tools*.

Also note that the compilers used by conda-build can be specified using a `conda_build_config.yaml`. For more information about how to do that, see *Using your customized compiler package with conda-build 3*.

Notice that this package has an `outputs` section. This section is a list of packages to output as a result of building this package. In this case, the packages libgdal and GDAL will be built. Similar to a normal recipe, the outputs can have build scripts, tests scripts and requirements specified. For more information on how outputs work, see the *Outputs section*.

Now, let's try to build GDAL against some build matrix. We will specify building against Python 3.7 and 3.5 using a conda-build config. Add the following to your `conda_build_config.yaml`:

```
python:
   - 3.7
   - 3.5
```

Now you can build GDAL using conda-build with the command:

```
conda build gdal-feedstock
```

Or explicitly set the location of the conda-build variant matrix:

```
conda build gdal-feedstock --variant-config-file conda_build_config.yaml
```

If you want to know more about build variants and `conda_build_config.yaml`, including how to specify a config file and what can go into it, take a look at *Creating conda-build variant config files*.

3.2.2 Building conda packages with conda skeleton

- *Overview*
- *Who is this for?*
- *Before you start*
- *Building a simple package with conda skeleton pypi*
- *Optional---Building for a different Python version*
- *Optional---Converting conda package for other platforms*
- *Optional---Uploading packages to Anaconda.org*
- *Troubleshooting a sample issue*
- *More information*

Overview

This tutorial describes how to quickly build a conda package for a Python module that is already available on PyPI.

In the first procedure, building a simple package, you build a package that can be installed in any conda environment of the same Python version as your root environment. The remaining optional procedures describe how to build packages for other Python versions and other architectures, as well as how to upload packages to your Anaconda.org account.

Note: You may consider using Docker to run the tutorial.

Who is this for?

This tutorial is for Windows, macOS, and Linux users who wish to build a conda package from a PyPI package. No prior knowledge of conda-build or conda recipes is required.

Before you start

Before you start, check the *Prerequisites*.

Building a simple package with conda skeleton pypi

The `conda skeleton` command picks up the PyPI package metadata and prepares the conda-build recipe. The final step is to build the package itself and install it into your conda environment.

It is easy to build a skeleton recipe for any Python package that is hosted on PyPI, the official third-party software repository for the Python programming language.

In this section you are going to use conda skeleton to generate a conda recipe, which informs conda-build about where the source files are located and how to build and install the package.

You'll be working with a package named Click that is hosted on PyPI. Click is a tool for exposing Python functions to create command line interfaces.

First, in your user home directory, run the `conda skeleton` command:

```
conda skeleton pypi click
```

The two arguments to `conda skeleton` are the hosting location, in this case `pypi`, and the name of the package.

This creates a directory named `click` and creates one skeleton file in that directory: `meta.yaml`. Many other files can be added there as necessary, such as `build.sh` and `bld.bat`, test scripts, or anything else you need to build your software. For simple, pure-Python recipes, these extra files are unnecessary and the build/script section in the `meta.yaml` is sufficient. Use the `ls` command on macOS or Linux or the `dir` command on Windows to verify that this file has been created. The `meta.yaml` file has been populated with information from the PyPI metadata and in many cases will not need to be edited.

Files in the folder with `meta.yaml` are collectively referred to as the "conda-build recipe":

- `meta.yaml`---Contains all the metadata in the recipe. Only the package name and package version sections are required---everything else is optional.
- `bld.bat`---Windows commands to build the package.
- `build.sh`---macOS and Linux commands to build the package.

Now that you have the conda-build recipe ready, you can use conda-build to create the package:

```
conda-build click
```

When conda-build is finished, it displays the exact path and filename of the conda package. See *Troubleshooting a sample issue* if the `conda-build` command fails.

Windows example file path:

```
C:\Users\jsmith\miniconda\conda-bld\win-64\click-7.0-py37_0.tar.bz2
```

macOS example file path:

```
/Users/jsmith/miniconda/conda-bld/osx-64/click-7.0-py37_0.tar.bz2
```

Linux example file path:

```
/home/jsmith/miniconda/conda-bld/linux-64/click-7.0-py37_0.tar.bz2
```

Note: Your path and filename will vary depending on your installation and operating system. Save the path and filename information for the next step.

Now you can install your newly built package in your conda environment by using the use-local flag:

```
conda install --use-local click
```

Notice that Click is coming from the local conda-build channel.

```
(click) 0561:~ jsmith$ conda list
# packages in environment at /Users/Jsmith/miniconda/envs/click:
# Name                    Version                   Build  Channel
ca-certificates           2019.1.23                     0
certifi                   2019.3.9                py37_0
click                     7.0                     py37_0    local
```

Now verify that Click installed successfully:

```
conda list
```

Scroll through the list until you find Click.

At this point you now have a conda package for Click that can be installed in any conda environment of the same Python version as your root environment. The remaining optional sections show you how to make packages for other Python versions and other architectures and how to upload them to your Anaconda.org account.

Optional---Building for a different Python version

By default, conda-build creates packages for the version of Python installed in the root environment. To build packages for other versions of Python, you use the `--python` flag followed by a version. For example, to explicitly build a version of the Click package for Python 2.7, use:

```
conda-build --python 2.7 click
```

Notice that the file printed at the end of the `conda-build` output has changed to reflect the requested version of Python. `conda install` will look in the package directory for the file that matches your current Python version.

Windows example file path:

```
C:\Users\jsmith\Miniconda\conda-bld\win-64\click-7.0-py27_0.tar.bz2
```

macOS example file path:

```
/Users/jsmith/miniconda/conda-bld/osx-64/click-7.0-py27_0.tar.bz2
```

Linux example file path:

```
/home/jsmith/miniconda/conda-bld/linux-64/click-7.0-py27_0.tar.bz2
```

Note: Your path and filename will vary depending on your installation and operating system. Save the path and filename information for the next task.

Optional---Converting conda package for other platforms

Now that you have built a package for your current platform with conda-build, you can convert it for use on other platforms with the `conda convert` command. This works only for pure Python packages where there is no compiled code. Conda convert does nothing to change compiled code, it only adapts file paths to take advantage of the fact that Python scripts are mostly platform independent. Conda convert accepts a platform specifier from this and a platform specifier from this list:

- osx-64.
- linux-32.
- linux-64.
- win-32.
- win-64.
- all.

In the output directory, 1 folder will be created for each of the 1 or more platforms you chose and each folder will contain a .tar.bz2 package file for that platform.

Windows:

```
conda convert -f --platform all C:\Users\jsmith\miniconda\conda-bld\win-64\click-7.0-
↪py37_0.tar.bz2
-o outputdir\
```

macOS and Linux:

```
conda convert --platform all /home/jsmith/miniconda/conda-bld/linux-64/click-7.0-py37_
↪0.tar.bz2
-o outputdir/
```

Note: Change your path and filename to the exact path and filename you saved in *Optional---Building for a different Python version*.

To use these packages, you need to transfer them to other computers and place them in the correct `conda-bld/$ARCH` directory for the platform, where `$ARCH` can be `osx-64`, `linux-32`, `linux-64`, `win-32`, or `win-64`.

A simpler way is to upload all of the bz2 files to Anaconda.org as described in the next task.

If you find yourself needing to use `conda convert`, you might instead prefer to change your recipe to make your package a "noarch" package. Noarch packages run anywhere and do not require conda convert. Some of the ecosystem tools don't yet support noarch packages but, for the most part, noarch packages are a better way to go.

Optional---Uploading packages to Anaconda.org

Anaconda.org is a repository for public or private packages. Uploading to Anaconda.org allows you to easily install your package in any environment with just the `conda install` command, rather than manually copying or moving the tarball file from one location to another. You can choose to make your files public or private. For more information about Anaconda.org, see the Anaconda.org documentation.

1. Create a free Anaconda.org account and record your new Anaconda.org username and password.
2. Run `conda install anaconda-client` and enter your Anaconda.org username and password.
3. Log into your Anaconda.org account from your terminal with the command `anaconda login`.

Now you can upload the new local packages to Anaconda.org.

Windows:

```
anaconda upload C:\Users\jsmith\miniconda\conda-bld\win-64\click-7.0-py37_0.tar.bz2
```

macOS and Linux:

```
anaconda upload /home/jsmith/miniconda/conda-bld/linux-64/click-7.0-py37_0.tar.bz2
```

Note: Change your path and filename to the exact path and filename you saved in *Optional---Building for a different Python version*. Your path and filename will vary depending on your installation and operating system.

If you created packages for multiple versions of Python or used `conda convert` to make packages for each supported architecture, you must use the `anaconda upload` command to upload each one. It is considered best practice to create packages for Python versions 2.7, 3.4, and 3.5 along with all of the architectures.

Tip: If you want to always automatically upload a successful build to Anaconda.org, run: `conda config --set anaconda_upload yes`

You can log out of your Anaconda.org account with the command:

```
anaconda logout
```

Troubleshooting a sample issue

Conda-build may produce the error message "Build Package missing."

To explore this error:

1. Create a conda skeleton package for skyfield. The `conda skeleton` command is:

   ```
   conda skeleton pypi skyfield
   ```

   This command creates the skyfield conda-build recipe.

2. Run `conda-build skyfield` and observe that it fails with the following output:

```
Removing old build environment
Removing old work directory
BUILD START: skyfield-0.8-py35_0
Using Anaconda Cloud api site https://api.anaconda.org
Fetching package metadata: ......
Solving package specifications: .
Error:  Package missing in current osx-64 channels:
  - sgp4 >=1.4
```

In this example, the conda recipe requires `sgp4` for the skyfield package. The skyfield recipe was created by `conda skeleton`. This error means that conda could not find the sgp4 package and install it.

Since many PyPI packages depend on other PyPI packages to build or run, the solution is sometimes as simple as using `conda skeleton` to create a conda recipe for the missing package and then building it:

```
conda skeleton sgp4
conda build sgp4
```

You may also try using the `--recursive` flag with `conda skeleton`, but this makes conda recipes for all required packages, even those that are already available to conda install.

More information

For more options, see the full *conda skeleton command documentation*.

3.2.3 Building conda packages from scratch

- *Overview*
- *Who is this for?*
- *Before you start*
- *Editing the meta.yaml file*
- *Writing the build script files build.sh and bld.bat*
- *Building and installing*
- *Converting a package for use on all platforms*
- *Optional---Using PyPI as the source instead of GitHub*
- *Optional---Uploading new packages to Anaconda.org*
- *More information*

Overview

This tutorial describes how to build a conda package for Click by writing the required files in the conda-build recipe.

Who is this for?

This tutorial is for Windows, macOS, and Linux users who wish to generate a conda package by writing the necessary files. Prior knowledge of conda-build and conda recipes is helpful.

Before you start

- Check the *prerequisites*.
- You should have already completed *Building conda packages with conda skeleton*.

Editing the meta.yaml file

1. Make a new directory for this tutorial named `click`, and then change to the new directory:

   ```
   mkdir click
   cd click
   ```

2. To create a new `meta.yaml` file, open your favorite editor. Create a new text file and insert the information shown below. A blank sample `meta.yaml` follows the table to make it easier to match up the information.

   Note: To allow correct sorting and comparison, specify `version` as a string.

name	click
version	"7.0" (or latest from https://github.com/pallets/click/releases)
git_rev	6.7 (or latest from https://github.com/pallets/click/releases)
git_url	https://github.com/pallets/click.git
imports	click
home	https://github.com/pallets/click
license	BSD

   ```
   package:
     name:
     version:

   source:
     git_rev:
     git_url:

   requirements:
     build:
       - python
       - setuptools

     run:
       - python

   ```

(continues on next page)

(continued from previous page)

```
test:
  imports:
    -

about:
  home:
```

3. Save the file in the same `click` directory as `meta.yaml`. It should match `this meta.yaml file`.

Writing the build script files build.sh and bld.bat

Besides `meta.yaml`, 2 files are required for a build:

- `build.sh`---Shell script for macOS and Linux.
- `bld.bat`---Batch file for Windows.

These 2 build files contain all the variables, such as for 32-bit or 64-bit architecture---the ARCH variable---and the build environment prefix---PREFIX. The 2 files `build.sh` and `bld.bat` must be in the same directory as your `meta.yaml` file.

This tutorial describes how to make both `build.sh` and `bld.bat` so that other users can build the appropriate package for their architecture.

1. Open a text editor and create a new file named `bld.bat`. Type the text exactly as shown:

```
"%PYTHON%" setup.py install
if errorlevel 1 exit 1
```

Note: In `bld.bat`, the best practice is to to add `if errorlevel 1 exit 1` after every command so that if the command fails, the build fails.

2. Save this new file `bld.bat` to the same directory where you put your `meta.yaml` file.
3. Open a text editor and create a new file named `build.sh`. Enter the text exactly as shown:

```
$PYTHON setup.py install     # Python command to install the script.
```

4. Save your new `build.sh` file to the same directory where you put the `meta.yaml` file.

You can run `build.sh` with `bash -x -e`. The `-x` makes it echo each command that is run, and the `-e` makes it exit whenever a command in the script returns nonzero exit status. If you need to revert this in the script, use the `set` command in `build.sh`.

Building and installing

Now that you have your 3 new build files ready, you are ready to create your new package with conda-build and install the package on your local computer.

1. Run conda-build:

```
conda-build click
```

If you are already in the click folder, you can type `conda build .`.

When conda-build is finished, it displays the package filename and location. In this case the file is saved to:

```
~/anaconda/conda-bld/linux-64/click-7.0-py37_0.tar.bz2
```

Note: Save this path and file information for the next task. The exact path and filename vary depending on your operating system and whether you are using Anaconda or Miniconda. The `conda-build` command tells you the exact path and filename.

2. Install your newly built program on your local computer by using the `use-local` flag:

   ```
   conda install --use-local click
   ```

   If there are no error messages, Click installed successfully.

   Note: Explicitly installing a local package bypasses the dependency resolver, as such the package's `run` dependencies will not be evaluated. See *conda install --help* or the install command reference page for more info.

Converting a package for use on all platforms

Now that you have built a package for your current platform with conda-build, you can convert it for use on other platforms by using the 2 build files, `build.sh` and `bld.bat`.

Use the `conda convert` command with a platform specifier from the list:

- `osx-64`.
- `linux-32`.
- `linux-64`.
- `win-32`.
- `win-64`.
- `all`.

EXAMPLE: Using the platform specifier `all`:

```
conda convert --platform all ~/anaconda/conda-bld/linux-64/click-7.0-py37_0.tar.bz2 -
↪o outputdir/
```

Note: Change your path and filename to the path and filename you saved in *Building and installing*.

Optional---Using PyPI as the source instead of GitHub

You can use PyPI or another repository instead of GitHub. There is little difference to conda-build between building from Git versus building from a tarball on a repository like PyPI. Because the same source is hosted on PyPI and GitHub, you can easily find a script on PyPI instead of GitHub.

Replace this `source` section:

```
git_rev: v0.6.7
git_url: https://github.com/pallets/click.git
```

With the following:

```
url: https://files.pythonhosted.org/packages/f8/5c/
↪f60e9d8a1e77005f664b76ff8aeaee5bc05d0a91798afd7f53fc998dbc47/Click-7.0.tar.gz
sha256: 5b94b49521f6456670fdb30cd82a4eca9412788a93fa6dd6df72c94d5a8ff2d7
```

Note: The `url` and `sha256` are found on the PyPI Click page.

Optional---Uploading new packages to Anaconda.org

After converting your files for use on other platforms, you may choose to upload your files to Anaconda.org, formerly known as binstar.org. It only takes a minute to do if you have a free Anaconda.org account.

1. If you have not done so already, open a free Anaconda.org account and record your new user name and password.
2. Run the command `conda install anaconda-client`, and then enter your Anaconda.org username and password.
3. Log into your Anaconda.org account with the command:

   ```
   anaconda login
   ```

4. Upload your package to Anaconda.org:

   ```
   anaconda upload ~/miniconda/conda-bld/linux-64/click-7.0-py37_0.tar.bz2
   ```

   Note: Change your path and filename to the path and filename you saved in *Building and installing*.

   Tip: To save time, you can set conda to always upload a successful build to Anaconda.org with the command: `conda config --set anaconda_upload yes`.

More information

- For more information about all the possible values that can go into the `meta.yaml` file, see *Defining metadata (meta.yaml)*.

3.2.4 Building R packages with skeleton CRAN

- *Overview*
- *Who is this for?*
- *Before you start*
- *Building a simple package with conda skeleton CRAN*
- *Building an R package with dependencies*
- *Optional---Building for a different R version*

- *Optional---Uploading packages to Anaconda.org*
- *More information*

Overview

This tutorial describes how to quickly build an R-language package on macOS for an R module that is already available on CRAN.

You will build a simple package that can be installed in any conda environment of the same R version as your root environment. The tutorial will also tell you how to build the dependencies that may arise while building the package.

Who is this for?

This tutorial is for macOS users who wish to build an R-language package from CRAN. No prior knowledge of conda-build or conda recipes is required.

Before you start

Before you start, check the *prerequisites*.

Building a simple package with conda skeleton CRAN

The conda skeleton command picks up the CRAN package metadata and prepares the conda-build recipe. The final step is to build the package itself and install it into your conda environment.

It is easy to build a skeleton recipe for any R package that is hosted on CRAN. In this section you are going to use conda skeleton to generate a conda recipe, which informs conda-build about where the source files are located and how to build and install the package.

You'll be working with a package that is hosted on CRAN named fansi, a tool that accounts for the effects of ANSI text formatting control sequences.

First, in your user home directory, run the conda skeleton command:

```
conda skeleton cran fansi
```

The two arguments to `conda skeleton` are the type of hosting location, in this case `cran`, and the name of the package.

This creates a directory named `r-fansi` and creates 1 skeleton file in that directory: `meta.yaml`. Many other files can be added there as necessary, such as `build.sh` and `bld.bat`, test scripts, or anything else you need to build your software. Use the `ls` command to verify that this file has been created. The `meta.yaml` file has been populated with information from the CRAN metadata and in many cases will not need to be edited.

Files in the folder with `meta.yaml` are collectively referred to as the "conda-build recipe":

- `meta.yaml`---Contains all the metadata in the recipe. The package name and package version sections are required---everything else is optional.
- `bld.bat`---Windows commands to build the package.
- `build.sh`---macOS and Linux commands to build the package.

Now that you have the conda-build recipe ready, you can use conda-build to create the package:

```
conda-build r-fansi
```

When conda-build is finished, it displays the exact path and filename of the conda package. See *Troubleshooting a sample issue* if the `conda-build` command fails. If you receive an error with SDK on macOS, review our *resources for macOS and SDK*.

Example file path:

```
/Users/jsmith/anaconda3/conda-bld/osx-64/r-fansi-0.4.0-r353h46e59ec_0.tar.bz2
```

Note: Your path and filename will vary depending on your installation and operating system. Save the path and filename information for the next step.

Now you can install your newly built package in your conda environment by using the use-local flag:

```
conda install --use-local r-fansi
```

Now verify that fansi installed successfully:

```
conda list
```

Scroll through the list until you find `r-fansi`.

Notice that fansi is coming from the local conda-build channel.

```
(base) 0561:~ jsmith$ conda list
# packages in environment at /Users/Jsmith/anaconda3:
# Name                    Version                   Build  Channel
qtpy                      1.5.0                    py37_0
r-base                    3.5.1                h539fb6c_1
r-fansi                   0.4.0              r353h46e59ec_0    local
```

The version of R will be what you have in your base environment.

See *Optional---Building for a different R version* to set your own R version.

At this point you now have a conda package for fansi that can be installed in any conda environment of its R version.

Building an R package with dependencies

The fansi package was a simple one that didn't have dependencies. To build an R package with dependencies, let's look at the example of janitor. Janitor is a package hosted on CRAN that is used for examining and cleaning up data.

To begin building it, type:

```
conda skeleton cran janitor
```

This creates a directory named `r-janitor` and creates one skeleton file in that directory: `meta.yaml`. Many other files can be added there as necessary, such as `build.sh` and `bld.bat`, test scripts, or anything else you need to build your software. Use the `ls` command to verify that this file has been created. The `meta.yaml` file has been populated with information from the CRAN metadata and in many cases will not need to be edited.

Now that you have the conda-build recipe ready, you can use conda-build to create the package:

```
conda-build r-janitor
```

What may happen at this point is that you will have dependencies of this package that do not exist as conda packages yet. They need to be turned into conda packages. Use conda skeleton to recursively build out recipes for the packages that it depends on:

```
conda skeleton cran janitor --recursive
```

You can manually build each package individually by typing:

```
conda-build package-name
```

Note: Replace "package-name" with the name of each package.

Once all of the package dependencies are resolved, you can build the R package by using:

```
conda-build .
```

Now you can install your newly built package in your conda environment by using the use-local flag:

```
conda install --use-local r-janitor
```

The remaining optional sections show you how to make R packages for other R versions and other architectures and how to upload them to your Anaconda.org account.

Optional---Building for a different R version

By default, conda-build creates packages for the version of R installed in the root environment. To build packages for other versions of R, you use the `--R` flag followed by a version.

For example, to explicitly build a version of the fansi package for R 3.6.1, use:

```
conda-build --R 3.6.1 r-fansi
```

Notice that the file printed at the end of the conda-build output has changed to reflect the requested version of R. Conda install will look in the package directory for the file that matches your current R version.

Example file path:

```
/Users/jsmith/anaconda3/conda-bld/osx-64/r-fansi-0.4.0-r353h46e59ec_0.tar.bz2
```

Note: Your path and filename will vary depending on your installation and operating system. Save the path and filename information for the next task.

Optional---Uploading packages to Anaconda.org

Anaconda.org is a repository for public or private packages. Uploading to Anaconda.org allows you to easily install your package in any environment with just the `conda install` command, rather than manually copying or moving the tarball file from one location to another. You can choose to make your files public or private.

For more information about Anaconda.org, see the Anaconda.org documentation.

1. Create a free Anaconda.org account and record your new Anaconda.org username and password.
2. Run `conda install anaconda-client` and enter your Anaconda.org username and password.

3. Log into your Anaconda.org account from your terminal with the command `anaconda login`.

Now you can upload the new local packages to Anaconda.org.

```
anaconda upload /Users/jsmith/anaconda3/conda-bld/osx-64/r-fansi-0.4.0-r353h46e59ec_0.
↪tar.bz2
```

Note: Change your path and filename to the exact path and filename you saved in *Optional---Building for a different R version*. Your path and filename will vary depending on your installation and operating system. If you created packages for multiple versions of R, you must use the `anaconda upload` command to upload each one.

Tip: If you want to always automatically upload a successful build to Anaconda.org, run: `conda config --set anaconda_upload yes`

You can log out of your Anaconda.org account with the command:

```
anaconda logout
```

More information

For more options, see the full *conda skeleton command documentation*.

3.3 Recipes

Review *recipe concepts* for more information about conda-build recipes.

3.3.1 Building a package without a recipe (bdist_conda)

- *Setup options*
  - *Build number*
  - *Build string*
  - *Import tests*
  - *Command line tests*
  - *Binary files relocatable*
  - *Preserve egg directory*
  - *Features*
  - *Track features*
- *Command line options*
  - *Build number*
- *Notes*

You can use conda-build to build packages for Python to install rather than conda by using `setup.py bdist_conda`. This is a quick way to build packages without using a recipe, but it has limitations. The script is limited to the Python version used in the build and it is not as reproducible as using a recipe. We recommend using a recipe with conda-build.

Note: If you use Setuptools, you must first import Setuptools and then import `distutils.command.bdist_conda`, because Setuptools monkey patches `distutils.dist.Distribution`.

EXAMPLE: A minimal `setup.py` file using the setup options `name` and `version`:

```
from distutils.core import setup, Extension
import distutils.command.bdist_conda

setup(
    name="foo",
    version="1.0",
    distclass=distutils.command.bdist_conda.CondaDistribution,
    conda_buildnum=1,
    conda_features=['mkl'],
)
```

Setup options

You can pass the following options to `setup()`. You must include `distclass=distutils.command.bdist_conda.CondaDistribution)`.

Build number

The number of the build. Can be overridden on the command line with the `--buildnum` flag. Defaults to `0`.

```
conda_buildnum=1
```

Build string

The build string. Default is generated automatically from the Python version, NumPy version---if relevant---and the build number, such as `py34_0`.

```
conda_buildstr=py34_0
```

Import tests

Whether to automatically run import tests. The default is `True`, which runs import tests for all the modules in `packages`. Also allowed are `False`, which runs no tests, or a list of module names to be tested on import.

```
conda_import_tests=False
```

Command line tests

Command line tests to run. Default is `True`, which runs `command --help` for each command in the console_scripts and gui_scripts entry_points. Also allowed are `False`, which does not run any command tests, or a list of command tests to run.

```
conda_command_tests=False
```

Binary files relocatable

Whether binary files should be made relocatable, using install_name_tool on macOS or patchelf on Linux. The default is `True`.

```
conda_binary_relocation=False
```

For more information, see *Making packages relocatable*.

Preserve egg directory

Whether to preserve the egg directory as installed by Setuptools. The default is `True` if the package depends on Setuptools or has Setuptools entry_points other than console_scripts and gui_scripts.

```
conda_preserve_egg_dir=False
```

Features

A list of features for the package.

```
conda_features=['mkl']
```

Note: Replace `mkl` with the features that you want to list.

Track features

List of features that this package should track---enable---when installed.

```
conda_track_features=['mkl']
```

Command line options

Build number

Set the build number. Defaults to the conda_buildnum passed to `setup()` or `0`. Overrides any conda_buildnum passed to `setup()`.

```
--buildnum=1
```

Notes

- You must install `bdist_conda` into a root conda environment, as it imports `conda` and `conda_build`. It is included as part of the `conda-build` package.
- All metadata is gathered from the standard metadata from the `setup()` function. Metadata that are not directly supported by `setup()` can be added using one of the options specified above.
- By default, import tests are run for each subpackage specified by packages, and command line tests `command --help` are run for each `setuptools entry_points` command. This is done to ensure that the package is built correctly. You can disable or change these using the `conda_import_tests` and `conda_command_tests` options specified above.
- The Python version used in the build must be the same as where conda is installed, as `bdist_conda` uses `conda-build`.
- `bdist_conda` uses the metadata provided to the `setup()` function.
- If you want to pass any `bdist_conda` specific options to `setup()`, in `setup()` you must set `distclass=distutils.command.bdist_conda.CondaDistribution`.

3.3.2 Sample recipes

Conda offers you the flexibility of being able to build things that are not Python related. The first 2 sample recipes, `boost` and `libtiff`, are examples of non-Python libraries, meaning they do not require Python to run or build.

- boost is an example of a popular programming library and illustrates the use of selectors in a recipe.
- libtiff is another example of a compiled library, which shows how conda can apply patches to source directories before building the package.
- msgpack, blosc, and cytoolz are examples of Python libraries with extensions.
- toolz, sympy, six, and gensim are examples of Python-only libraries.

`gensim` works on Python 2, and all of the others work on both Python 2 and Python 3.

3.3.3 Debugging conda recipes

Recipes are something that you'll rarely get exactly right on the first try. Something about the build will be wrong, and the build will break. Maybe you only notice a problem during tests, but you need more info than you got from the tests running in conda-build. Conda-build 3.17.0 added the subcommand, `conda debug`, that is designed to facilitate the recipe debugging process.

Fundamentally, debugging is a process of getting into or recreating the environment and set of shell environment variables that conda-build creates during its build or test processes. This has been possible for a very long time---you could observe the build output, figure out where the files from your build were placed, navigate there, and finally, activate the appropriate environment(s). Then you might also need to set some environment variables manually.

What `conda debug` does is to create environments for you and provide you with a single command line that you can copy/paste to enter a debugging environment.

Usage

The `conda debug` command accepts 1 of 2 kinds of inputs: a recipe folder or a path to a built package.

If a path to a recipe folder is provided, `conda debug` creates the build and host environments. It provisions any source code that your recipe specifies. It leaves the build-time scripts in the work folder for you. When complete, `conda debug` prints something like this:

```
################################################################################
Build and/or host environments created for debugging.  To enter a debugging␣
↪environment:

cd /Users/UserName/miniconda3/conda-bld/debug_1542385789430/work && source /Users/
↪UserName/miniconda3/conda-bld/debug_1542385789430/work/build_env_setup.sh

To run your build, you might want to start with running the conda_build.sh file.
################################################################################
```

If a path to a built package is provided, `conda debug` creates the test environment. It prepares any test files that the recipe specified. When complete, `conda debug` prints something like this:

```
################################################################################
Test environment created for debugging.  To enter a debugging environment:

cd /Users/UserName/miniconda3/conda-bld/conda-build_1542302975704/work && source /
↪Users/UserName/miniconda3/conda-bld/conda-build_1542302975704/work/build_env_setup.
↪sh

To run your tests, you might want to start with running the conda_test_runner.sh file.
################################################################################
```

Next steps

Given the output above, you can now enter an environment to start debugging. Copy paste from your terminal and go:

```
cd /Users/UserName/miniconda3/conda-bld/debug_1542385789430/work && source /Users/
↪UserName/miniconda3/conda-bld/debug_1542385789430/work/build_env_setup.sh
```

This is where you'll hopefully know what build commands you want to run to help you debug. Every build is different so your experience will vary. However, if you have no idea at all, you could probably start by running the appropriate build or test script, as mentioned in the output. If you do this, remember that these scripts might be written to exit on error, which may close your shell session. It may be wise to only run these scripts in an explicit subshell:

```
bash conda_build.sh
bash conda_test_runner.sh
```

Complications with multiple outputs

Multiple outputs effectively give the recipe many build phases to consider. The `--output-id` argument is the mechanism to specify which of these should be used to create the debug envs and scripts. The `--output-id` argument accepts an fnmatch pattern. You can match any part of the output filenames. This really only works for conda packages, not other output types, such as wheels, because conda-build can't currently predict their filenames without actually carrying out a build.

For example, our NumPy recipe has multiple outputs. If we wanted to debug the NumPy-base output, we would specify it with a command like:

```
conda debug numpy-feedstock --output-id="numpy-base*"
```

If you have a matrix build, you may need to be more specific:

```
Specified --output-id matches more than one output (['/Users/msarahan/miniconda3/
↪conda-bld/debug_1542387301945/osx-64/numpy-base-1.14.6-py27h1a60bec_4.tar.bz2', '/
↪Users/msarahan/miniconda3/conda-bld/debug_1542387301945/osx-64/numpy-base-1.14.6-
↪py27h8a80b8c_4.tar.bz2', '/Users/msarahan/miniconda3/conda-bld/debug_1542387301945/
↪osx-64/numpy-base-1.14.6-py36h1a60bec_4.tar.bz2', '/Users/msarahan/miniconda3/conda-
↪bld/debug_1542387301945/osx-64/numpy-base-1.14.6-py36h8a80b8c_4.tar.bz2', '/Users/
↪msarahan/miniconda3/conda-bld/debug_1542387301945/osx-64/numpy-base-1.14.6-
↪py37h1a60bec_4.tar.bz2', '/Users/msarahan/miniconda3/conda-bld/debug_1542387301945/
↪osx-64/numpy-base-1.14.6-py37h8a80b8c_4.tar.bz2']).  Please refine your output id␣
↪so that only a single output is found.
```

You could either reduce your matrix by changing your `conda_build_config.yaml`, or making a simpler one and passing it on the CLI, or by using the CLI to reduce it.

```
conda debug numpy-feedstock --output-id="numpy-base*" --python=3.6 --variants="{blas_
↪impl: openblas}"
```

```
Specified --output-id matches more than one output (['/Users/UserName/miniconda3/
↪conda-bld/debug_1542387443190/osx-64/numpy-base-1.14.6-py36h28eea48_4.tar.bz2', '/
↪Users/UserName/miniconda3/conda-bld/debug_1542387443190/osx-64/numpy-base-1.14.6-
↪py36ha711998_4.tar.bz2']).  Please refine your output id so that only a single␣
↪output is found.
```

However, this is still not enough as our matrix includes two BLAS implementations, MKL and OpenBLAS.

Further reduction:

```
conda debug numpy-feedstock --output-id="numpy-base*" --python=3.6 --variants="{blas_
↪impl: 'openblas'}"
```

Cleanup

Debugging folders are named in a way that the `conda build purge` command will find and clean up. If you use the -p/--path CLI argument, conda-build will not detect these and you'll need to manually clean up yourself. `conda build purge-all` will also remove previously built packages.

Quirks

You can specify where you want the root of your debugging stuff to go with the -p/--path CLI argument. The way this works is that conda-build treats that as its "croot" where packages get cached as necessary, as well as potentially indexed. When using the --path argument, you may see folders like "osx-64" or other platform subdirs in the path you specify. It is safe to remove them or ignore them.

3.4 Environment variables

- *Dynamic behavior based on state of build process*
- *Environment variables set during the build process*
- *Git environment variables*
- *Mercurial environment variables*
- *Inherited environment variables*
- *Environment variables that affect the build process*
- *Environment variables to set build features*
- *Environment variables that affect the test process*

3.4.1 Dynamic behavior based on state of build process

There are times when you may want to process a single file in different ways at more than 1 step in the render-build-test flow of conda-build. Conda-build sets the CONDA_BUILD_STATE environment variable during each of these phases. The possible values are:

- RENDER---Set during evaluation of the `meta.yaml` file.
- BUILD---Set during processing of the `bld.bat` or `build.sh` script files.
- TEST---Set during the running of any `run_test` scripts, which also includes any commands defined in `meta.yaml` in the `test/commands` section.

The CONDA_BUILD_STATE variable is undefined outside of these locations.

3.4.2 Environment variables set during the build process

During the build process, the following environment variables are set, on Windows with `bld.bat` and on macOS and Linux with `build.sh`. By default, these are the only variables available to your build script. Unless otherwise noted, no variables are inherited from the shell environment in which you invoke `conda-build`. To override this behavior, see *Inherited environment variables*.

ARCH	Either `32` or `64`, to specify whether the build is 32-bit or 64-bit. The value depends on the ARCH environment variable and defaults to the architecture the interpreter running conda was compiled with.
CMAKE_GENERATOR	The CMake generator string for the current build environment. On Linux systems, this is always `Unix Makefiles`. On Windows, it is generated according to the Visual Studio version activated at build time, for example, `Visual Studio 9 2008 Win64`.
CONDA_BUILD=1	Always set.
CPU_COUNT	The number of CPUs on the system, as reported by `multiprocessing.cpu_count()`.
SHLIB_EXT	The shared library extension.
DIRTY	Set to 1 if the `--dirty` flag is passed to the `conda-build` command. May be used to skip parts of a build script conditionally for faster iteration time when developing recipes. For example, downloads, extraction and other things that need not be repeated.
HTTP_PROXY	Inherited from your shell environment.
HTTPS_PROXY	Inherited from your shell environment.
LANG	Inherited from your shell environment.
MAKEFLAGS	Inherited from your shell environment. May be used to set additional arguments to make, such as `-j2`, which uses 2 CPU cores to build your recipe.
PY_VER	Python version building against. Set with the `--python` argument or with the CONDA_PY environment variable.
NPY_VER	NumPy version to build against. Set with the `--numpy` argument or with the CONDA_NPY environment variable.
PATH	Inherited from your shell environment and augmented with `$PREFIX/bin`.
PREFIX	Build prefix to which the build script should install.
PKG_BUILDNUM	Build number of the package being built.
PKG_NAME	Name of the package being built.
PKG_VERSION	Version of the package being built.
`PKG_BUILD_STRING`	Complete build string of the package being built, including hash. EXAMPLE: py27h21422ab_0 . Conda-build 3.0+.
`PKG_HASH`	Hash of the package being built, without leading h. EXAMPLE: 21422ab . Conda-build 3.0+.
PYTHON	Path to the Python executable in the host prefix. Python is installed only in the host prefix when it is listed as a host requirement.
PY3K	`1` when Python 3 is installed in the build prefix, otherwise `0`.
R	Path to the R executable in the build prefix. R is only installed in the build prefix when it is listed as a build requirement.
RECIPE_DIR	Directory of the recipe.
SP_DIR	Python's site-packages location.
SRC_DIR	Path to where source is unpacked or cloned. If the source file is not a recognized file type---zip, tar, tar.bz2, or tar.xz---this is a directory containing a copy of the source file.
STDLIB_DIR	Python standard library location.

Unix-style packages on Windows, which are usually statically linked to executables, are built in a special `Library` directory under the build prefix. The environment variables listed in the following table are defined only on Windows.

CYGWIN_PREFIX	Same as PREFIX, but as a Unix-style path, such as `/cygdrive/c/path/to/prefix`.
LIBRARY_BIN	`<build prefix>\Library\bin`.
LIBRARY_INC	`<build prefix>\Library\include`.
LIBRARY_LIB	`<build prefix>\Library\lib`.
LIBRARY_PREFIX	`<build prefix>\Library`.
SCRIPTS	`<build prefix>\Scripts`.
VS_MAJOR	The major version number of the Visual Studio version activated within the build, such as `9`.
VS_VERSION	The version number of the Visual Studio version activated within the build, such as `9.0`.
VS_YEAR	The release year of the Visual Studio version activated within the build, such as `2008`.

The environment variables listed in the following table are defined only on macOS and Linux.

HOME	Standard $HOME environment variable.
PKG_CONFIG_PATH	Path to `pkgconfig` directory.

The environment variables listed in the following table are defined only on macOS.

CFLAGS	`-arch` flag.
CXXFLAGS	Same as CFLAGS.
LDFLAGS	Same as CFLAGS.
MACOSX_DEPLOYMENT_TARGET	Same as the Anaconda Python macOS deployment target. Currently `10.9`.
OSX_ARCH	`i386` or `x86_64`, depending on Python build.

The environment variable listed in the following table is defined only on Linux.

LD_RUN_PATH	`<build prefix>/lib`.

3.4.3 Git environment variables

The environment variables listed in the following table are defined when the source is a git repository, specifying the source either with git_url or path.

GIT_BUILD_STR	String that joins GIT_DESCRIBE_NUMBER and GIT_DESCRIBE_HASH by an underscore.
GIT_DESCRIBE_HASH	The current commit short-hash as displayed from `git describe --tags`.
GIT_DESCRIBE_NUMBER	String denoting the number of commits since the most recent tag.
GIT_DESCRIBE_TAG	String denoting the most recent tag from the current commit, based on the output of `git describe --tags`.
GIT_FULL_HASH	String with the full SHA1 of the current HEAD.

These can be used in conjunction with templated `meta.yaml` files to set things---such as the build string---based on the state of the git repository.

3.4.4 Mercurial environment variables

The environment variables listed in the following table are defined when the source is a mercurial repository.

HG_BRANCH	String denoting the presently active branch.
HG_BUILD_STR	String that joins HG_NUM_ID and HG_SHORT_ID by an underscore.
HG_LATEST_TAG	String denoting the most recent tag from the current commit.
HG_LATEST_TAG_DISTANCE	String denoting number of commits since the most recent tag.
HG_NUM_ID	String denoting the revision number.
HG_SHORT_ID	String denoting the hash of the commit.

3.4.5 Inherited environment variables

Other than those mentioned above, no variables are inherited from the environment in which you invoke conda-build. You can choose to inherit additional environment variables by adding them to `meta.yaml`:

```
build:
  script_env:
   - TMPDIR
   - LD_LIBRARY_PATH # [linux]
   - DYLD_LIBRARY_PATH # [osx]
```

If an inherited variable is missing from your shell environment, it remains unassigned, but a warning is issued noting that it has no value assigned.

Warning: Inheriting environment variables can make it difficult for others to reproduce binaries from source with your recipe. Use this feature with caution or avoid it.

Note: If you split your build and test phases with `--no-test` and `--test`, you need to ensure that the environment variables present at build time and test time match. If you do not, the package hashes may use different values and your package may not be testable because the hashes will differ.

3.4.6 Environment variables that affect the build process

CONDA_PY	The Python version used to build the package. Should be `27`, `34`, `35`, `36`, or `37`.
CONDA_NPY	The NumPy version used to build the package, such as `19`, `110`, or `111`.
CONDA_PREFIX	The path to the conda environment used to build the package, such as `/path/to/conda/env`. Useful to pass as the environment prefix parameter to various conda tools, usually labeled `-p` or `--prefix`.

3.4.7 Environment variables to set build features

The environment variables listed in the following table are inherited from the process running conda-build.

FEATURE_NOMKL	Adds the `nomkl` feature to the built package.	Accepts `0` for off and `1` for on.
FEATURE_DEBUG	Adds the `debug` feature to the built package.	Accepts `0` for off and `1` for on.
FEATURE_OPT	Adds the `opt` feature to the built package.	Accepts `0` for off and `1` for on.

3.4.8 Environment variables that affect the test process

All of the above environment variables are also set during the test process, using the test prefix instead of the build prefix.

3.5 Using wheel files with conda

If you have software in a Python wheel file and want to use it with conda or install it in a conda environment, there are 3 ways.

The best way is to obtain the source code for the software and build a conda package from the source and not from a wheel. This helps ensure that the new package uses other conda packages to satisfy its dependencies.

The second best way is to build a conda package from the wheel file. This tells conda more about the files present than a pip install. It is also less likely than a pip install to cause errors by overwriting (or "clobbering") files. Building a conda package from the wheel file also has the advantage that any clobbering is more likely to happen at build time and not runtime.

The third way is to use pip to install a wheel file into a conda environment. Some conda users have used this option safely. The first 2 ways are still the safest and most reliable.

3.5.1 Building a conda package from a wheel file

To build a conda package from a wheel file, install the .whl file in the conda recipe's `bld.bat` or `build.sh` file.

You may download the .whl file in the source section of the conda recipe's `meta.yaml` file.

You may instead put the URL directly in the `pip install` command.

EXAMPLE: The conda recipe for TensorFlow has a `pip install` command in build.sh with the URL of a .whl file. The meta.yaml file does not download or list the .whl file.

Note: It is important to `pip install` only the one desired package. Whenever possible, install dependencies with conda and not pip.

We strongly recommend using the `--no-deps` option in the `pip install` command.

If you run `pip install` without the `--no-deps` option, pip will often install dependencies in your conda recipe and those dependencies will become part of your package. This wastes space in the package and increases the risk of file overlap, file clobbering, and broken packages.

Tutorials

The *tutorials* will guide you through how to build conda packages---whether you're creating a package with compilers, using conda skeleton, creating from scratch, or building R packages using skeleton CRAN.

Recipes

Conda-build uses *recipes* to create conda packages. We have guides on debugging conda recipes, sample recipes for you to use, and information on how to build a package without a recipe.

Environment variables

Use our *environment variables* guide to understand which environment variables are available, set, and inherited, and how they affect different processes.

Wheel files

The user guide includes information about *wheel files* and how to build conda packages from them.

CHAPTER FOUR

RESOURCES

These resources will help you accomplish more using conda-build. We provide information on topics including build scripts, build variants, making packages relocatable, and defining metadata in the meta.yaml. We also provide guidelines and a template for submitting your own documentation.

4.1 Build scripts (build.sh, bld.bat)

The `build.sh` file is the build script for Linux and macOS and `bld.bat` is the build script for Windows. These scripts contain the logic that carries out your build steps. Traditionally it has also included install steps. With the traditional one-package-per-recipe way of doing things, anything that your build script copies into the `$PREFIX` or `%PREFIX%` folder will be included in your output package. For example, this `build.sh`:

```
mkdir -p $PREFIX/bin
cp $RECIPE_DIR/my_script_with_recipe.sh $PREFIX/bin/super-cool-script.sh
```

If you don't care about deploying your package with pip on PyPI, this can save you a lot of time in figuring out the proper way to include additional files with setup.py.

There are many environment variables defined for you to use in build.sh and bld.bat. Please see *Environment variables* for more information.

As of conda-build 2.1, you can also define multiple output packages. Each package has its own script or list of files to include. The rules for these outputs are documented at *Outputs section*. When any output is defined, this overrides the default behavior of bundling anything in `$PREFIX`. So to output multiple packages from a single recipe, remove any installation steps from `build.sh` or `bld.bat` and do them instead in your install script(s) for each output.

`build.sh` and `bld.bat` are optional. You can instead use the `build/script` key in your `meta.yaml`, with each value being either a string command or a list of string commands. Any commands you put there must be able to run on every platform for which you build. For example, you can't use the `cp` command because cmd.exe won't understand it in Windows.

`build.sh` is run with `bash` and `bld.bat` is run with `cmd.exe`.

There is some development towards the ability to use bash scripts in Windows, but this is not currently supported. You may write your script as a .sh file, and then call it in your bld.bat file, but there is no way to directly run build.sh on Windows. The conda recipe at https://github.com/AnacondaRecipes/conda-feedstock/tree/master/recipe is an example of this method.

4.2 Anaconda compiler tools

Anaconda 5.0 switched from OS-provided compiler tools to our own toolsets. This allows improved compiler capabilities, including better security and performance. This page describes how to use these tools and enable these benefits.

4.2.1 Compiler packages

Before Anaconda 5.0, compilers were installed using system tools such as XCode or `yum install gcc`. Now there are conda packages for Linux and macOS compilers. Unlike the previous GCC 4.8.5 packages that included GCC, g++, and GFortran all in the same package, these conda packages are split into separate compilers:

macOS:

- clang_osx-64.
- clangxx_osx-64.
- gfortran_osx-64.

Linux:

- gcc_linux-64.
- gxx_linux-64.
- gfortran_linux-64.

A compiler's "build platform" is the platform where the compiler runs and builds the code.

A compiler's "host platform" is the platform where the built code will finally be hosted and run.

Notice that all of these package names end in a platform identifier which specifies the host platform. All compiler packages are specific to both the build platform and the host platform.

4.2.2 Using the compiler packages

The compiler packages can be installed with conda. Because they are designed with (pseudo) cross-compiling in mind, all of the executables in a compiler package are "prefixed." Instead of `gcc`, the executable name of the compiler you use will be something like `x86_64-conda_cos6-linux-gnu-gcc`. These full compiler names are shown in the build logs, recording the host platform and helping prevent the common mistake of using the wrong compiler.

Many build tools such as `make` and `CMake` search by default for a compiler named simply `gcc`, so we set environment variables to point these tools to the correct compiler.

We set these variables in conda `activate.d` scripts, so any environment in which you will use the compilers must first be activated so the scripts will run. Conda-build does this activation for you using activation hooks installed with the compiler packages in `CONDA_PREFIX/etc/conda/activate.d`, so no additional effort is necessary.

You can activate the root environment with the command `conda activate root`.

4.2.3 macOS SDK

The macOS compilers require the macOS 10.9 SDK. The SDK license prevents it from being bundled in the conda package. We know of 2 current sources for the macOS 10.9 SDK:

- https://github.com/devernay/xcodelegacy
- https://github.com/phracker/MacOSX-SDKs

We usually install this SDK at `/opt/MacOSX10.9.sdk` but you may install it anywhere. Edit your `conda_build_config.yaml` file to point to it, like this:

```
CONDA_BUILD_SYSROOT:
  - /opt/MacOSX10.9.sdk        # [osx]
```

At Anaconda, we have this configuration setting in a centralized `conda_build_config.yaml` at the root of our recipe repository. Since we run build commands from that location, the file and the setting are used for all recipes. The `conda_build_config.yaml` search order is described further at *Creating conda-build variant config files*.

Build scripts for macOS should make use of the variables `MACOSX_DEPLOYMENT_TARGET` and `CONDA_BUILD_SYSROOT`, which are set by conda-build (see *Environment variables*). These variables should be translated into correct compiler arguments, e.g. for Clang this would be:

```
clang .. -isysroot ${CONDA_BUILD_SYSROOT} -mmacosx-version-min=${MACOSX_DEPLOYMENT_
↪TARGET} ..
```

Most build tools, e.g. CMake and distutils (setuptools), will automatically pick up `MACOSX_DEPLOYMENT_TARGET` but you need to pass `CONDA_BUILD_SYSROOT` explicitly. For CMake, this can be done with the option `-DCMAKE_OSX_SYSROOT=${CONDA_BUILD_SYSROOT}`. When building Python extensions with distutils, one should always extend `CFLAGS` before calling `setup.py`:

```
export CFLAGS="${CFLAGS} -i sysroot ${CONDA_BUILD_SYSROOT}"
```

When building C++ extensions with Cython, `CXXFLAGS` must be similarly modified.

4.2.4 Backward compatibility

Some users want to use the latest Anaconda packages but do not yet want to use the Anaconda compilers. To enable this, the latest Python package builds have a default `_sysconfigdata` file. This file sets the compilers provided by the system, such as `gcc` and `g++`, as the default compilers. This way allows legacy recipes to keep working.

Python packages also include an alternative `_sysconfigdata` file that sets the Anaconda compilers as the default compilers. The Anaconda Python executable itself is made with these Anaconda compilers.

The compiler packages set the environment variable `_PYTHON_SYSCONFIGDATA_NAME`, which tells Python which `_sysconfigdata` file to use. This variable is set at activation time using the activation hooks described above.

The new `_sysconfigdata` customization system is only present in recent versions of the Python package. Conda-build automatically tries to use the latest Python version available in the currently configured channels, which normally gets the latest from the default channel. If you're using something other than conda-build while working with the new compilers, conda does not automatically update Python, so make sure you have the correct `_sysconfigdata` files by updating your Python package manually.

4.2.5 Anaconda compilers and conda-build 3

The Anaconda 5.0 compilers and conda-build 3 are designed to work together.

Conda-build 3 defines a special jinja2 function, `compiler()`, to make it easy to specify compiler packages dynamically on many platforms. The `compiler` function takes at least 1 argument, the language of the compiler to use:

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

"Cross-capable" recipes can be used to make packages with a host platform different than the build platform where conda-build runs. To write cross-capable recipes, you may also need to use the "host" section in the requirements section. In this example we set "host" to "zlib" to tell conda-build to use the zlib in the conda environment and not the system zlib. This makes sure conda-build uses the zlib for the host platform and not the zlib for the build platform.

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

Generally, the build section should include compilers and other build tools and the host section should include everything else, including shared libraries, Python, and Python libraries.

4.2.6 An aside on CMake and sysroots

Anaconda's compilers for Linux are built with something called crosstool-ng. They include not only GCC, but also a "sysroot" with glibc, as well as the rest of the toolchain (binutils). Ordinarily, the sysroot is something that your system provides, and it is what establishes the libc compatibility bound for your compiled code. Any compilation that uses a sysroot other than the system sysroot is said to be "cross-compiling." When the target OS and the build OS are the same, it is called a "pseudo-cross-compiler." This is the case for normal builds with Anaconda's compilers on Linux.

Unfortunately, some software tools do not handle sysroots in intuitive ways. CMake is especially bad for this. Even though the compiler itself understands its own sysroot, CMake insists on ignoring that. We've filed issues at:

- https://gitlab.kitware.com/cmake/cmake/issues/17483

Additionally, this Stack Overflow issue has some more information: https://stackoverflow.com/questions/36195791/cmake-missing-sysroot-when-cross-compiling

In order to teach CMake about the sysroot, you must do additional work. As an example, please see our recipe for libnetcdf at https://github.com/AnacondaRecipes/libnetcdf-feedstock/tree/master/recipe

In particular, you'll need to copy the `cross-linux.cmake` file there, and reference it in your build.sh file:

```
CMAKE_PLATFORM_FLAGS+=(-DCMAKE_TOOLCHAIN_FILE="${RECIPE_DIR}/cross-linux.cmake")

cmake -DCMAKE_INSTALL_PREFIX=${PREFIX} \
  ${CMAKE_PLATFORM_FLAGS[@]} \
  ${SRC_DIR}
```

4.2.7 Customizing the compilers

The compiler packages listed above are small packages that only include the activation scripts and list most of the software they provide as runtime dependencies.

This design is intended to make it easy for you to customize your own compiler packages by copying these recipes and changing the flags. You can then edit the `conda_build_config.yaml` file to specify your own packages.

We have been careful to select good, general purpose, secure, and fast flags. We have also used them for all packages in Anaconda Distribution 5.0.0, except for some minor customizations in a few recipes. When changing these flags, remember that choosing the wrong flags can reduce security, reduce performance, and cause incompatibilities.

With that warning in mind, let's look at good ways to customize Clang.

1. Download or fork the code from https://github.com/anacondarecipes/aggregate. The Clang package recipe is in the `clang` folder. The main material is in the llvm-compilers-feedstock folder.
2. Edit `clang/recipe/meta.yaml`:

   ```
   package:
     name: clang_{{ target_platform }}
     version: {{ version }}
   ```

   The name here does not matter but the output names below do. Conda-build expects any compiler to follow the BASENAME_PLATFORMNAME pattern, so it is important to keep the `{{target_platform}}` part of the name.

   `{{ version }}` is left as an intentionally undefined jinja2 variable. It is set later in `conda_build_config.yaml`.
3. Before any packaging is done, run the build.sh script: https://github.com/AnacondaRecipes/aggregate/blob/master/clang/build.sh

   In this recipe, values are changed here. Those values are inserted into the activate scripts that are installed later.

   ```
   #!/bin/bash

   CHOST=${macos_machine}

   FINAL_CPPFLAGS="-D_FORTIFY_SOURCE=2 -mmacosx-version-min=${macos_min_version}"
   FINAL_CFLAGS="-march=core2 -mtune=haswell -mssse3 -ftree-vectorize -fPIC -fPIE -
   ↪fstack-protector-strong -O2 -pipe"
   FINAL_CXXFLAGS="-march=core2 -mtune=haswell -mssse3 -ftree-vectorize -fPIC -fPIE -
   ↪fstack-protector-strong -O2 -pipe -stdlib=libc++ -fvisibility-inlines-hidden -
   ↪std=c++14 -fmessage-length=0"
   # These are the LDFLAGS for when the linker is being called directly, without "-
   ↪Wl,"
   FINAL_LDFLAGS="-pie -headerpad_max_install_names"
   # These are the LDFLAGS for when the linker is being driven by a compiler, with "-
   ↪Wl,"
   FINAL_LDFLAGS_CC="-Wl,-pie -Wl,-headerpad_max_install_names"
   FINAL_DEBUG_CFLAGS="-Og -g -Wall -Wextra -fcheck=all -fbacktrace -fimplicit-none -
   ↪fvar-tracking-assignments"
   FINAL_DEBUG_CXXFLAGS="-Og -g -Wall -Wextra -fcheck=all -fbacktrace -fimplicit-
   ↪none -fvar-tracking-assignments"
   FINAL_DEBUG_FFLAGS="-Og -g -Wall -Wextra -fcheck=all -fbacktrace -fimplicit-none -
   ↪fvar-tracking-assignments"

   find "${RECIPE_DIR}" -name "*activate*.sh" -exec cp {} . \;
   ```

(continues on next page)

(continued from previous page)

```
find . -name "*activate*.sh" -exec sed -i.bak "s|@CHOST@|${CHOST}|g" "{}" \;
find . -name "*activate*.sh" -exec sed -i.bak "s|@CPPFLAGS@|${FINAL_CPPFLAGS}|g" ␣
↪           "{}" \;
find . -name "*activate*.sh" -exec sed -i.bak "s|@CFLAGS@|${FINAL_CFLAGS}|g"     ␣
↪           "{}" \;
find . -name "*activate*.sh" -exec sed -i.bak "s|@DEBUG_CFLAGS@|${FINAL_DEBUG_
↪CFLAGS}|g"     "{}" \;
find . -name "*activate*.sh" -exec sed -i.bak "s|@CXXFLAGS@|${FINAL_CXXFLAGS}|g" ␣
↪           "{}" \;
find . -name "*activate*.sh" -exec sed -i.bak "s|@DEBUG_CXXFLAGS@|${FINAL_DEBUG_
↪CXXFLAGS}|g" "{}" \;
find . -name "*activate*.sh" -exec sed -i.bak "s|@DEBUG_CXXFLAGS@|${FINAL_DEBUG_
↪CXXFLAGS}|g" "{}" \;
# find . -name "*activate*.sh" -exec sed -i.bak "s|@FFLAGS@|${FINAL_FFLAGS}|g"   ␣
↪             "{}" \;
# find . -name "*activate*.sh" -exec sed -i.bak "s|@DEBUG_FFLAGS@|${FINAL_DEBUG_
↪FFLAGS}|g"     "{}" \;
find . -name "*activate*.sh" -exec sed -i.bak "s|@LDFLAGS@|${FINAL_LDFLAGS}|g"   ␣
↪           "{}" \;
find . -name "*activate*.sh" -exec sed -i.bak "s|@LDFLAGS_CC@|${FINAL_LDFLAGS_CC}
↪|g"         "{}" \;
find . -name "*activate*.sh.bak" -exec rm "{}" \;
```

4. With those changes to the activate scripts in place, it's time to move on to installing things. Look back at the `clang` folder's `meta.yaml`. Here's where we change the package name. Notice what comes before the `{{ target_platform }}`.

```
outputs:
  - name: super_duper_clang_{{ target_platform }}
    script: install-clang.sh
    requirements:
      - clang {{ version }}
```

The script reference here is another place you might add customization. You'll either change the contents of those install scripts or change the scripts that those install scripts are installing.

Note that we make the package `clang` in the main material agree in version with our output version. This is implicitly the same as the top-level recipe. The `clang` package sets no environment variables at all, so it may be difficult to use directly.

5. Let's examine the script `install-clang.sh`:

```
#!/bin/bash

set -e -x

CHOST=${macos_machine}

mkdir -p "${PREFIX}"/etc/conda/{de,}activate.d/
cp "${SRC_DIR}"/activate-clang.sh "${PREFIX}"/etc/conda/activate.d/activate_"$
↪{PKG_NAME}".sh
cp "${SRC_DIR}"/deactivate-clang.sh "${PREFIX}"/etc/conda/deactivate.d/deactivate_
↪"${PKG_NAME}".sh

pushd "${PREFIX}"/bin
  ln -s clang ${CHOST}-clang
popd
```

Nothing here is too unusual.

Activate scripts are named according to our package name so they won't conflict with other activate scripts.

The symlink for Clang is a Clang implementation detail that sets the host platform.

We define `macos_machine` in aggregate's `conda_build_config.yaml`: https://github.com/AnacondaRecipes/aggregate/blob/master/conda_build_config.yaml#L79

The activate scripts that are being installed are where we actually set the environment variables. Remember that these have been modified by build.sh.

6. With any of your desired changes in place, go ahead and build the recipe.

   You should end up with a super_duper_clang_osx-64 package. Or, if you're not on macOS and are modifying a different recipe, you should end up with an equivalent package for your platform.

4.2.8 Using your customized compiler package with conda-build 3

Remember the Jinja2 function, `{{ compiler('c') }}`? Here's where that comes in. Specific keys in `conda_build_config.yaml` are named for the language argument to that jinja2 function. In your `conda_build_config.yaml`, add this:

```
c_compiler:
  - super_duper_clang
```

Note that we're not adding the `target_platform` part, which is separate. You can define that key, too:

```
c_compiler:
  - super_duper_clang
target_platform:
  - win-64
```

With those two keys defined, conda-build will try to use a compiler package named `super_duper_clang_win-64`. That package needs to exist for your native platform. For example, if you're on macOS, your native platform is `osx-64`.

The package subdirectory for your native platform is the build platform. The build platform and the `target_platform` can be the same, and they are the same by default, but they can also be different. When they are different, you're cross-compiling.

If you ever needed a different compiler key for the same language, remember that the language key is arbitrary. For example, we might want different compilers for Python and for R within one ecosystem. On Windows, the Python ecosystem uses the Microsoft Visual C compilers, while the R ecosystem uses the Mingw compilers.

Let's start in `conda_build_config.yaml`:

```
python_c_compiler:
  - vs2015
r_c_compiler:
  - m2w64-gcc
target_platform:
  - win-64
```

In Python recipes, you'd have:

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

In R recipes, you'd have:

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

This example is a little contrived, because the `m2w64-gcc_win-64` package is not available. You'd need to create a metapackage `m2w64-gcc_win-64` to point at the `m2w64-gcc` package, which does exist on the msys2 channel on repo.anaconda.com.

4.2.9 Anaconda compilers implicitly add RPATH pointing to the conda environment

You might want to use the Anaconda compilers outside of `conda-build` so that you use the same versions, flags, and configuration, for maximum compatibility with Anaconda packages (but in a case where you want simple tarballs, for example). In this case, there is a gotcha.

Even if Anaconda compilers are used from outside of `conda-build`, the GCC specs are customized so that, when linking an executable or a shared library, an RPATH pointing to `lib/` inside the current enviroment prefix directory (`$CONDA_PREFIX/lib`) is added. This is done by changing the `link_libgcc:` section inside GCC `specs` file, and this change is done so that `LD_LIBRARY_PATH` isn't required for basic libraries.

`conda-build` knows how to make this automatically relocatable, so that this `RPATH` will be changed to point to the environment where the package is being installed (at installation time, by `conda`). But if you only pack this binary in a tarball, it will continue containing this hardcoded `RPATH` to an environment in your machine. In this case, it is recommended to manually remove the `RPATH`.

4.3 Defining metadata (meta.yaml)

- *Package section*
- *Source section*
- *Build section*
- *Requirements section*
- *Test section*
- *Outputs section*
- *About section*
- *App section*
- *Extra section*
- *Templating with Jinja*
- *Preprocessing selectors*

All the metadata in the conda-build recipe is specified in the `meta.yaml` file. See the example below:

```
{% set version = "1.1.0" %}

package:
  name: imagesize
```

(continues on next page)

(continued from previous page)

```
  version: {{ version }}

source:
  url: https://pypi.io/packages/source/i/imagesize/imagesize-{{ version }}.tar.gz
  sha256: f3832918bc3c66617f92e35f5d70729187676313caa60c187eb0f28b8fe5e3b5

build:
  noarch: python
  number: 0
  script: python -m pip install --no-deps --ignore-installed .

requirements:
  host:
    - python
    - pip
  run:
    - python

test:
  imports:
    - imagesize

about:
  home: https://github.com/shibukawa/imagesize_py
  license: MIT
  summary: 'Getting image size from png/jpeg/jpeg2000/gif file'
  description: |
    This module analyzes jpeg/jpeg2000/png/gif image header and
    return image size.
  dev_url: https://github.com/shibukawa/imagesize_py
  doc_url: https://pypi.python.org/pypi/imagesize
  doc_source_url: https://github.com/shibukawa/imagesize_py/blob/master/README.rst
```

All sections are optional except for `package/name` and `package/version`.

Headers must appear only once. If they appear multiple times, only the last is remembered. For example, the `package:` header should appear only once in the file.

4.3.1 Package section

Specifies package information.

Package name

The lower case name of the package. It may contain "-", but no spaces.

```
package:
  name: bsdiff4
```

Package version

The version number of the package. Use the PEP-386 verlib conventions. Cannot contain "-". YAML interprets version numbers such as 1.0 as floats, meaning that 0.10 will be the same as 0.1. To avoid this, put the version number in quotes so that it is interpreted as a string.

```
package:
  version: "1.1.4"
```

Note: Post-build versioning: In some cases, you may not know the version, build number, or build string of the package until after it is built. In these cases, you can perform *Templating with Jinja* or utilize *Git environment variables* and *Inherited environment variables*.

4.3.2 Source section

Specifies where the source code of the package is coming from. The source may come from a tarball file, git, hg, or svn. It may be a local path and it may contain patches.

Source from tarball or zip archive

```
source:
  url: https://pypi.python.org/packages/source/b/bsdiff4/bsdiff4-1.1.4.tar.gz
  md5: 29f6089290505fc1a852e176bd276c43
  sha1: f0a2c9a30073449cfb7d171c57552f3109d93894
  sha256: 5a022ff4c1d1de87232b1c70bde50afbb98212fd246be4a867d8737173cf1f8f
```

If an extracted archive contains only 1 folder at its top level, its contents will be moved 1 level up, so that the extracted package contents sit in the root of the work folder.

Source from git

The git_url can also be a relative path to the recipe directory.

```
source:
  git_url: https://github.com/ilanschnell/bsdiff4.git
  git_rev: 1.1.4
  git_depth: 1 # (Defaults to -1/not shallow)
```

The depth argument relates to the ability to perform a shallow clone. A shallow clone means that you only download part of the history from Git. If you know that you only need the most recent changes, you can say, `git_depth: 1`, which is faster than cloning the entire repo. The downside to setting it at 1 is that, unless the tag is on that specific commit, then you won't have that tag when you go to reference it in `git_rev` (for example). If your `git_depth` is insufficient to capture the tag in `git_rev`, you'll encounter an error. So in the example above, unless the 1.1.4 is the very head commit and the one that you're going to grab, you may encounter an error.

Source from hg

```yaml
source:
  hg_url: ssh://hg@bitbucket.org/ilanschnell/bsdiff4
  hg_tag: 1.1.4
```

Source from svn

```yaml
source:
  svn_url: https://github.com/ilanschnell/bsdiff
  svn_rev: 1.1.4
  svn_ignore_externals: True # (defaults to False)
```

Source from a local path

If the path is relative, it is taken relative to the recipe directory. The source is copied to the work directory before building.

```yaml
source:
  path: ../src
```

If the local path is a git or svn repository, you get the corresponding environment variables defined in your build environment. The only practical difference between git_url or hg_url and path as source arguments is that git_url and hg_url would be clones of a repository, while path would be a copy of the repository. Using path allows you to build packages with unstaged and uncommitted changes in the working directory. git_url can build only up to the latest commit.

Patches

Patches may optionally be applied to the source.

```yaml
source:
  #[source information here]
  patches:
    - my.patch # the patch file is expected to be found in the recipe
```

Conda-build automatically determines the patch strip level.

Destination path

Within conda-build's work directory, you may specify a particular folder to place source into. This feature is new in conda-build 3.0. Conda-build will always drop you into the same folder (build folder/work), but it's up to you whether you want your source extracted into that folder, or nested deeper. This feature is particularly useful when dealing with multiple sources, but can apply to recipes with single sources as well.

```yaml
source:
  #[source information here]
  folder: my-destination/folder
```

Filename

The filename key is `fn`. It was formerly required with URL source types. It is not required now.

If the `fn` key is provided, the file is saved on disk with that name. If the `fn` key is not provided, the file is saved on disk with a name matching the last part of the URL.

For example, `http://www.something.com/myfile.zip` has an implicit filename of `myfile.zip`. Users may change this by manually specifying `fn`.

```
source:
  url: http://www.something.com/myfile.zip
  fn: otherfilename.zip
```

Source from multiple sources

Some software is most easily built by aggregating several pieces. For this, conda-build 3.0 has added support for arbitrarily specifying many sources.

The syntax is a list of source dictionaries. Each member of this list follows the same rules as the single source for earlier conda-build versions (listed above). All features for each member are supported.

Example:

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

Here, the two URL tarballs will go into one folder, and the git repo is checked out into its own space. Git will not clone into a non-empty folder.

Note: Dashes denote list items in YAML syntax.

4.3.3 Build section

Specifies build information.

Each field that expects a path can also handle a glob pattern. The matching is performed from the top of the build environment, so to match files inside your project you can use a pattern similar to the following one: "**/myproject/**/*.txt". This pattern will match any .txt file found in your project.

Note: The quotation marks ("") are required for patterns that start with a *.

Recursive globbing using ** is supported only in conda-build >= 3.0.

Build number and string

The build number should be incremented for new builds of the same version. The number defaults to `0`. The build string cannot contain "-". The string defaults to the default conda-build string plus the build number.

```yaml
build:
  number: 1
  string: abc
```

A hash will appear when the package is affected by one or more variables from the conda_build_config.yaml file. The hash is made up from the "used" variables - if anything is used, you have a hash. If you don't use these variables then you won't have a hash. There are a few special cases that do not affect the hash, such as Python and R or anything that already had a place in the build string.

The build hash will be added to the build string if these are true for any dependency:

- package is an explicit dependency in build, host, or run deps
- package has a matching entry in conda_build_config.yaml which is a pin to a specific version, not a lower bound
- that package is not ignored by ignore_version

OR

- package uses {{ compiler() }} jinja2 function

Python entry points

The following example creates a Python entry point named "bsdiff4" that calls `bsdiff4.cli.main_bsdiff4()`.

```yaml
build:
  entry_points:
    - bsdiff4 = bsdiff4.cli:main_bsdiff4
    - bspatch4 = bsdiff4.cli:main_bspatch4
```

Python.app

If osx_is_app is set, entry points use `python.app` instead of Python in macOS. The default is `False`.

```yaml
build:
  osx_is_app: True
```

Track features

Adding track_features to one or more of the options will cause conda to de-prioritize it or “weigh it down.” The lowest priority package is the one that would cause the most track_features to be activated in the environment. The default package among many variants is the one that would cause the least track_features to be activated.

No two packages in a given subdir should ever have the same track_feature.

```yaml
build:
  track_features:
    - feature2
```

Preserve Python egg directory

This is needed for some packages that use features specific to setuptools. The default is `False`.

```
build:
  preserve_egg_dir: True
```

Skip compiling some .py files into .pyc files

Some packages ship `.py` files that cannot be compiled, such as those that contain templates. Some packages also ship `.py` files that should not be compiled yet, because the Python interpreter that will be used is not known at build time. In these cases, conda-build can skip attempting to compile these files. The patterns used in this section do not need the ** to handle recursive paths.

```
build:
  skip_compile_pyc:
    - "*/templates/*.py"          # These should not (and cannot) be compiled
    - "*/share/plugins/gdb/*.py"  # The python embedded into gdb is unknown
```

No link

A list of globs for files that should always be copied and never soft linked or hard linked.

```
build:
  no_link:
    - bin/*.py  # Don't link any .py files in bin/
```

Script

Used instead of `build.sh` or `bld.bat`. For short build scripts, this can be more convenient. You may need to use *selectors* to use different scripts for different platforms.

```
build:
  script: python setup.py install --single-version-externally-managed --record=record.
↪txt
```

RPATHs

Set which RPATHs are used when making executables relocatable on Linux. This is a Linux feature that is ignored on other systems. The default is `lib/`.

```
build:
  rpaths:
    - lib/
    - lib/R/lib/
```

Force files

Force files to always be included, even if they are already in the environment from the build dependencies. This may be needed, for example, to create a recipe for conda itself.

```
build:
  always_include_files:
    - bin/file1
    - bin/file2
```

Relocation

Advanced features. You can use the following 4 keys to control relocatability files from the build environment to the installation environment:

- binary_relocation.
- has_prefix_files.
- binary_has_prefix_files.
- ignore_prefix_files.

For more information, see *Making packages relocatable*.

Binary relocation

Whether binary files should be made relocatable using install_name_tool on macOS or patchelf on Linux. The default is `True`. It also accepts `False`, which indicates no relocation for any files, or a list of files, which indicates relocation only for listed files.

```
build:
  binary_relocation: False
```

Detect binary files with prefix

Binary files may contain the build prefix and need it replaced with the install prefix at installation time. Conda can automatically identify and register such files. The default is `True`.

Note: The default changed from `False` to `True` in conda build 2.0. Setting this to `False` means that binary relocation---RPATH---replacement will still be done, but hard-coded prefixes in binaries will not be replaced. Prefixes in text files will still be replaced.

```
build:
  detect_binary_files_with_prefix: False
```

Windows handles binary prefix replacement very differently than Unix-like systems such as macOS and Linux. At this time, we are unaware of any executable or library that uses hardcoded embedded paths for locating other libraries or program data on Windows. Instead, Windows follows DLL search path rules or more natively supports relocatability using relative paths. Because of this, conda ignores most prefixes. However, pip creates executables for Python entry points that do use embedded paths on Windows. Conda-build thus detects prefixes in all files and records them by default. If you are getting errors about path length on Windows, you should try to disable detect_binary_files_with_prefix. Newer versions of Conda, such as recent 4.2.x series releases and up, should have no problems here, but earlier versions of conda do erroneously try to apply any binary prefix replacement.

Binary has prefix files

By default, conda-build tries to detect prefixes in all files. You may also elect to specify files with binary prefixes individually. This allows you to specify the type of file as binary, when it may be incorrectly detected as text for some reason. Binary files are those containing NULL bytes.

```
build:
  binary_has_prefix_files:
    - bin/binaryfile1
    - lib/binaryfile2
```

Text files with prefix files

Text files---files containing no NULL bytes---may contain the build prefix and need it replaced with the install prefix at installation time. Conda will automatically register such files. Binary files that contain the build prefix are generally handled differently---see *Binary has prefix files*---but there may be cases where such a binary file needs to be treated as an ordinary text file, in which case they need to be identified.

```
build:
  has_prefix_files:
    - bin/file1
    - lib/file2
```

Ignore prefix files

Used to exclude some or all of the files in the build recipe from the list of files that have the build prefix replaced with the install prefix.

To ignore all files in the build recipe, use:

```
build:
  ignore_prefix_files: True
```

To specify individual filenames, use:

```
build:
  ignore_prefix_files:
    - file1
```

This setting is independent of RPATH replacement. Use the *Detect binary files with prefix* setting to control that behavior.

Skipping builds

Specifies whether conda-build should skip the build of this recipe. Particularly useful for defining recipes that are platform specific. The default is `False`.

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

Architecture independent packages

Allows you to specify "no architecture" when building a package, thus making it compatible with all platforms and architectures. Noarch packages can be installed on any platform.

Starting with conda-build 2.1, and conda 4.3, there is a new syntax that supports different languages. Assigning the noarch key as `generic` tells conda to not try any manipulation of the contents.

```
build:
  noarch: generic
```

`noarch: generic` is most useful for packages such as static javascript assets and source archives. For pure Python packages that can run on any Python version, you can use the `noarch: python` value instead:

```
build:
  noarch: python
```

The legacy syntax for `noarch_python` is still valid, and should be used when you need to be certain that your package will be installable where conda 4.3 is not yet available. All other forms of noarch packages require conda >=4.3 to install.

```
build:
  noarch_python: True
```

Warning: At the time of this writing, `noarch` packages should not make use of *preprocess-selectors*: `noarch` packages are built with the directives which evaluate to `True` in the platform it was built, which probably will result in incorrect/incomplete installation in other platforms.

Include build recipe

The full conda-build recipe and rendered `meta.yaml` file is included in the *Package metadata* by default. You can disable this with:

```
build:
  include_recipe: False
```

Use environment variables

Normally the build script in `build.sh` or `bld.bat` does not pass through environment variables from the command line. Only environment variables documented in *Environment variables* are seen by the build script. To "white-list" environment variables that should be passed through to the build script:

```
build:
  script_env:
    - MYVAR
    - ANOTHER_VAR
```

If a listed environment variable is missing from the environment seen by the conda-build process itself, a UserWarning is emitted during the build process and the variable remains undefined.

Note: Inheriting environment variables can make it difficult for others to reproduce binaries from source with your recipe. Use this feature with caution or avoid it.

Note: If you split your build and test phases with `--no-test` and `--test`, you need to ensure that the environment variables present at build time and test time match. If you do not, the package hashes may use different values, and your package may not be testable, because the hashes will differ.

Export runtime requirements

Some build or host *Requirements section* will impose a runtime requirement. Most commonly this is true for shared libraries (e.g. libpng), which are required for linking at build time, and for resolving the link at run time. With `run_exports` (new in conda-build 3) such a runtime requirement can be implicitly added by host requirements (e.g. libpng exports libpng), and with `run_exports/strong` even by build requirements (e.g. GCC exports libgcc).

```
# meta.yaml of libpng
build:
  run_exports:
    - libpng
```

Here, because no specific kind of `run_exports` is specified, libpng's `run_exports` are considered "weak." This means they will only apply when libpng is in the host section, when they will add their export to the run section. If libpng were listed in the build section, the `run_exports` would not apply to the run section.

```
# meta.yaml of gcc compiler
build:
  run_exports:
    strong:
      - libgcc
```

Strong `run_exports` are used for things like runtimes, where the same runtime needs to be present in the host and the run environment, and exactly which runtime that should be is determined by what's present in the build section. This mechanism is how we line up appropriate software on Windows, where we must match MSVC versions used across all of the shared libraries in an environment.

```
# meta.yaml of some package using gcc and libpng
requirements:
  build:
    - gcc            # has a strong run export
```

(continues on next page)

(continued from previous page)

```
  host:
    - libpng         # has a (weak) run export
    # - libgcc       <-- implicitly added by gcc
  run:
    # - libgcc       <-- implicitly added by gcc
    # - libpng       <-- implicitly added by libpng
```

You can express version constraints directly, or use any of the Jinja2 helper functions listed at *Extra Jinja2 functions*.

For example, you may use *Pinning expressions* to obtain flexible version pinning relative to versions present at build time:

```
build:
  run_exports:
    - {{ pin_subpackage('libpng', max_pin='x.x') }}
```

With this example, if libpng were version 1.6.34, this pinning expression would evaluate to `>=1.6.34,<1.7`.

Note that `run_exports` can be specified both in the build section and on a per-output basis for split packages.

`run_exports` only affects directly named dependencies. For example, if you have a metapackage that includes a compiler that lists `run_exports`, you also need to define `run_exports` in the metapackage so that it takes effect when people install your metapackage. This is important, because if `run_exports` affected transitive dependencies, you would see many added dependencies to shared libraries where they are not actually direct dependencies. For example, Python uses bzip2, which can use `run_exports` to make sure that people use a compatible build of bzip2. If people list python as a build time dependency, bzip2 should only be imposed for Python itself and should not be automatically imposed as a runtime dependency for the thing using Python.

The potential downside of this feature is that it takes some control over constraints away from downstream users. If an upstream package has a problematic `run_exports` constraint, you can ignore it in your recipe by listing the upstream package name in the `build/ignore_run_exports` section:

```
build:
  ignore_run_exports:
    - libstdc++
```

Pin runtime dependencies

The `pin_depends` build key can be used to enforce pinning behavior on the output recipe or built package.

There are 2 possible behaviors:

```
build:
  pin_depends: record
```

With a value of `record`, conda-build will record all requirements exactly as they would be installed in a file called info/requires. These pins will not show up in the output of `conda render` and they will not affect the actual run dependencies of the output package. It is only adding in this new file.

```
build:
  pin_depends: strict
```

With a value of `strict`, conda-build applies the pins to the actual metadata. This does affect the output of `conda render` and also affects the end result of the build. The package dependencies will be strictly pinned down to the build string level. This will supersede any dynamic or compatible pinning that conda-build may otherwise be doing.

Whitelisting shared libraries

The `missing_dso_whitelist` build key is a list of globs for dynamic shared object (DSO) files that should be ignored when examining linkage information.

During the post-build phase, the shared libraries in the newly created package are examined for linkages which are not provided by the package's requirements or a predefined list of system libraries. If such libraries are detected, either a warning `--no-error-overlinking` or error `--error-overlinking` will result.

```
build:
  missing_dso_whitelist:
```

These keys allow additions to the list of allowed libraries.

The `runpath_whitelist` build key is a list of globs for paths which are allowed to appear as runpaths in the package's shared libraries. All other runpaths will cause a warning message to be printed during the build.

```
build:
  runpath_whitelist:
```

4.3.4 Requirements section

Specifies the build and runtime requirements. Dependencies of these requirements are included automatically.

Versions for requirements must follow the conda match specification. See *Package match specifications*.

Build

Tools required to build the package. These packages are run on the build system and include things such as revision control systems (Git, SVN) make tools (GNU make, Autotool, CMake) and compilers (real cross, pseudo-cross, or native when not cross-compiling), and any source pre-processors.

Packages which provide "sysroot" files, like the `CDT` packages (see below) also belong in the build section.

```
requirements:
  build:
    - git
    - cmake
```

Host

This section was added in conda-build 3.0. It represents packages that need to be specific to the target platform when the target platform is not necessarily the same as the native build platform. For example, in order for a recipe to be "cross-capable", shared libraries requirements must be listed in the host section, rather than the build section, so that the shared libraries that get linked are ones for the target platform, rather than the native build platform. You should also include the base interpreter for packages that need one. In other words, a Python package would list `python` here and an R package would list `mro-base` or `r-base`.

```
requirements:
  build:
    - {{ compiler('c') }}
    - {{ cdt('xorg-x11-proto-devel') }}  # [linux]
  host:
    - python
```

Note: When both build and host sections are defined, the build section can be thought of as "build tools" - things that run on the native platform, but output results for the target platform. For example, a cross-compiler that runs on linux-64, but targets linux-armv7.

The PREFIX environment variable points to the host prefix. With respect to activation during builds, both the host and build environments are activated. The build prefix is activated before the host prefix so that the host prefix has priority over the build prefix. Executables that don't exist in the host prefix should be found in the build prefix.

As of conda-build 3.1.4, the build and host prefixes are always separate when both are defined, or when `{{ compiler() }}` Jinja2 functions are used. The only time that build and host are merged is when the host section is absent, and no `{{ compiler() }}` Jinja2 functions are used in meta.yaml. Because these are separate, you may see some build failures when migrating your recipes. For example, let's say you have a recipe to build a Python extension. If you add the compiler Jinja2 functions to the build section, but you do not move your Python dependency from the build section to the host section, your recipe will fail. It will fail because the host environment is where new files are detected, but because you have Python only in the build environment, your extension will be installed into the build environment. No files will be detected. Also, variables such as PYTHON will not be defined when Python is not installed into the host environment.

On Linux, using the compiler packages provided by Anaconda Inc. in the `defaults` meta-channel can prevent your build system leaking into the built software by using our `CDT` (Core Dependency Tree) packages for any "system" dependencies. These packages are repackaged libraries and headers from CentOS6 and are unpacked into the sysroot of our pseudo-cross compilers and are found by them automatically.

Note that what qualifies as a "system" dependency is a matter of opinion. The Anaconda Distribution chose not to provide X11 or GL packages, so we use CDT packages for X11. Conda-forge chose to provide X11 and GL packages.

On macOS, you can also use `{{ compiler() }}` to get compiler packages provided by Anaconda Inc. in the `defaults` meta-channel. The environment variables `MACOSX_DEPLOYMENT_TARGET` and `CONDA_BUILD_SYSROOT` will be set appropriately by conda-build (see *Environment variables*). `CONDA_BUILD_SYSROOT` will specify a folder containing a macOS SDK. These settings achieve backwards compatibility while still providing access to C++14 and C++17. Note that conda-build will set `CONDA_BUILD_SYSROOT` by parsing the `conda_build_config.yaml`. For more details, see *Anaconda compiler tools*.

TL;DR: If you use `{{ compiler() }}` Jinja2 to utilize our new compilers, you must also move anything that is not strictly a build tool into your host dependencies. This includes Python, Python libraries, and any shared libraries that you need to link against in your build. Examples of build tools include any `{{ compiler() }}`, Make, Autoconf, Perl (for running scripts, not installing Perl software), and Python (for running scripts, not for installing software).

Run

Packages required to run the package. These are the dependencies that are installed automatically whenever the package is installed. Package names should follow the package match specifications.

```
requirements:
  run:
    - python
    - argparse # [py26]
    - six >=1.8.0
```

To build a recipe against different versions of NumPy and ensure that each version is part of the package dependencies, list `numpy x.x` as a requirement in `meta.yaml` and use `conda-build` with a NumPy version option such as `--numpy 1.7`.

The line in the `meta.yaml` file should literally say `numpy x.x` and should not have any numbers. If the `meta.yaml` file uses `numpy x.x`, it is required to use the `--numpy` option with `conda-build`.

```
requirements:
  run:
    - python
    - numpy x.x
```

Note: Instead of manually specifying run requirements, since conda-build 3 you can augment the packages used in your build and host sections with *run_exports* which are then automatically added to the run requirements for you.

Run_constrained

Packages that are optional at runtime but must obey the supplied additional constraint if they are installed.

Package names should follow the package match specifications.

```
requirements:
  run_constrained:
    - optional-subpackage =={{ version }}
```

4.3.5 Test section

If this section exists or if there is a `run_test.[py,pl,sh,bat]` file in the recipe, the package is installed into a test environment after the build is finished and the tests are run there.

Test files

Test files that are copied from the recipe into the temporary test directory and are needed during testing.

```
test:
  files:
    - test-data.txt
```

Source files

Test files that are copied from the source work directory into the temporary test directory and are needed during testing.

```
test:
  source_files:
    - test-data.txt
    - some/directory
    - some/directory/pattern*.sh
```

This capability was added in conda-build 2.0.

Test requirements

In addition to the runtime requirements, you can specify requirements needed during testing. The runtime requirements that you specified in the "run" section described above are automatically included during testing.

```
test:
  requires:
    - nose
```

Test commands

Commands that are run as part of the test.

```
test:
  commands:
    - bsdiff4 -h
    - bspatch4 -h
```

Python imports

List of Python modules or packages that will be imported in the test environment.

```
test:
  imports:
    - bsdiff4
```

This would be equivalent to having a `run_test.py` with the following:

```
import bsdiff4
```

Run test script

The script `run_test.sh`---or `.bat`, `.py`, or `.pl`---is run automatically if it is part of the recipe.

Note: Python .py and Perl .pl scripts are valid only as part of Python and Perl packages, respectively.

Downstream tests

Knowing that your software built and ran its tests successfully is necessary, but not sufficient, for keeping whole systems of software running. To have confidence that a new build of a package hasn't broken other downstream software, conda-build supports the notion of downstream testing.

```
test:
  downstreams:
    - some_downstream_pkg
```

This is saying "When I build this recipe, after you run my test suite here, also download and run some_downstream_pkg which depends on my package." Conda-build takes care of ensuring that the package you just built gets installed into the environment for testing some_downstream_pkg. If conda-build can't create that environment due to unsatisfiable dependencies, it will skip those downstream tests and warn you. This usually happens when you are building a new version of a package that will require you to rebuild the downstream dependencies.

Downstreams specs are full conda specs, similar to the requirements section. You can put version constraints on your specs in here:

```
test:
  downstreams:
    - some_downstream_pkg  >=2.0
```

More than one package can be specified to run downstream tests for:

```
test:
  downstreams:
    - some_downstream_pkg
    - other_downstream_pkg
```

However, this does not mean that these packages are tested together. Rather, each of these are tested for satisfiability with your new package, then each of their test suites are run separately with the new package.

4.3.6 Outputs section

Explicitly specifies packaging steps. This section supports multiple outputs, as well as different package output types. The format is a list of mappings. Build strings for subpackages are determined by their runtime dependencies. This support was added in conda-build 2.1.0.

```
outputs:
  - name: some-subpackage
    version: 1.0
  - name: some-other-subpackage
    version: 2.0
```

Note: If any output is specified in the outputs section, the default packaging behavior of conda-build is bypassed. In other words, if any subpackage is specified, then you do not get the normal top-level build for this recipe without explicitly defining a subpackage for it. This is an alternative to the existing behavior, not an addition to it. For more information, see *Implicit metapackages*. Each output may have its own version and requirements. Additionally, subpackages may impose downstream pinning similarly to *Pin downstream* to help keep your packages aligned.

Specifying files to include in output

You can specify files to be included in the package in 1 of 2 ways:

- Explicit file lists.
- Scripts that move files into the build prefix.

Explicit file lists are relative paths from the root of the build prefix. Explicit file lists support glob expressions. Directory names are also supported, and they recursively include contents.

```
outputs:
  - name: subpackage-name
    files:
      - a-file
      - a-folder
```

(continues on next page)

(continued from previous page)

```
      - *.some-extension
      - somefolder/*.some-extension
```

Scripts that create or move files into the build prefix can be any kind of script. Known script types need only specify the script name. Currently the list of recognized extensions is py, bat, ps1, and sh.

```
outputs:
  - name: subpackage-name
    script: move-files.py
```

The interpreter command must be specified if the file extension is not recognized.

```
outputs:
  - name: subpackage-name
    script: some-script.extension
    script_interpreter: program plus arguments to run script
```

For scripts that move or create files, a fresh copy of the working directory is provided at the start of each script execution. This ensures that results between scripts are independent of one another.

Note: For either the file list or the script approach, having more than 1 package contain a given file is not explicitly forbidden, but may prevent installation of both packages simultaneously. Conda disallows this condition because it creates ambiguous runtime conditions.

Subpackage requirements

Like a top-level recipe, a subpackage may have zero or more dependencies listed as build requirements and zero or more dependencies listed as run requirements.

The dependencies listed as subpackage build requirements are available only during the packaging phase of that subpackage.

A subpackage does not automatically inherit any dependencies from its top-level recipe, so any build or run requirements needed by the subpackage must be explicitly specified.

```
outputs:

  - name: subpackage-name
    requirements:
      build:
        - some-dep
      run:
        - some-dep
```

It is also possible for a subpackage requirements section to have a list of dependencies, but no build section or run section. This is the same as having a build section with this dependency list and a run section with the same dependency list.

```
outputs:
  - name: subpackage-name
    requirements:
      - some-dep
```

You can also impose runtime dependencies whenever a given (sub)package is installed as a build dependency. For example, if we had an overarching "compilers" package, and within that, had `gcc` and `libgcc` outputs, we could force recipes that use GCC to include a matching libgcc runtime requirement:

```
outputs:
  - name: gcc
    run_exports:
      - libgcc 2.*
  - name: libgcc
```

See the *Export runtime requirements* section for additional information.

Note: Variant expressions are very powerful here. You can express the version requirement in the `run_exports` entry as a Jinja function to insert values based on the actual version of libgcc produced by the recipe. Read more about them at *Referencing subpackages*.

Implicit metapackages

When viewing the top-level package as a collection of smaller subpackages, it may be convenient to define the top-level package as a composition of several subpackages. If you do this and you do not define a subpackage name that matches the top-level package/name, conda-build creates a metapackage for you. This metapackage has runtime requirements drawn from its dependency subpackages, for the sake of accurate build strings.

EXAMPLE: In this example, a metapackage for `subpackage-example` will be created. It will have runtime dependencies on `subpackage1`, `subpackage2`, `some-dep`, and `some-other-dep`.

```
package:
  name: subpackage-example
  version: 1.0

requirements:
  run:
    - subpackage1
    - subpackage2

outputs:
  - name: subpackage1
    requirements:
      - some-dep
  - name: subpackage2
    requirements:
      - some-other-dep
  - name: subpackage3
    requirements:
      - some-totally-exotic-dep
```

Subpackage tests

You can test subpackages independently of the top-level package. Independent test script files for each separate package are specified under the subpackage's test section. These files support the same formats as the top-level `run_test.*` scripts, which are .py, .pl, .bat, and .sh. These may be extended to support other script types in the future.

```yaml
outputs:
  - name: subpackage-name
    test:
      script: some-other-script.py
```

By default, the `run_test.*` scripts apply only to the top-level package. To apply them also to subpackages, list them explicitly in the script section:

```yaml
outputs:
  - name: subpackage-name
    test:
      script: run_test.py
```

Test requirements for subpackages are not supported. Instead, subpackage tests install their runtime requirements---but not the run requirements for the top-level package---and the test-time requirements of the top-level package.

EXAMPLE: In this example, the test for `subpackage-name` installs `some-test-dep` and `subpackage-run-req`, but not `some-top-level-run-req`.

```yaml
requirements:
  run:
    - some-top-level-run-req

test:
  requires:
    - some-test-dep

outputs:
  - name: subpackage-name
    requirements:
      - subpackage-run-req
    test:
      script: run_test.py
```

Output type

Conda-build supports creating packages other than conda packages. Currently that support includes only wheels, but others may come as demand appears. If type is not specified, the default value is `conda`.

```yaml
requirements:
  build:
    - wheel

outputs:
  - name: name-of-wheel-package
    type: wheel
```

Currently you must include the wheel package in your top-level requirements/build section in order to build wheels.

When specifying type, the name field is optional and it defaults to the package/name field for the top-level recipe.

```yaml
requirements:
  build:
    - wheel

outputs:
  - type: wheel
```

Conda-build currently knows how to test only conda packages. Conda-build does support using Twine to upload packages to PyPI. See the conda-build help output (`conda-build --help`) for the list of arguments accepted that will be passed through to Twine.

Note: You must use pip to install Twine in order for this to work.

4.3.7 About section

Specifies identifying information about the package. The information displays in the Anaconda.org channel.

```yaml
about:
  home: https://github.com/ilanschnell/bsdiff4
  license: BSD
  license_file: LICENSE
  summary: binary diff and patch using the BSDIFF4-format
```

License file

Add a file containing the software license to the package metadata. Many licenses require the license statement to be distributed with the package. The filename is relative to the source directory. The value can be a single filename or a YAML list for multiple license files.

```yaml
about:
  license_file: LICENSE
```

4.3.8 App section

If the app section is present, the package is an app, meaning that it appears in Anaconda Navigator.

Entry point

The command that is called to launch the app in Navigator.

```yaml
app:
  entry: ipython notebook
```

Icon file

The icon file contained in the recipe.

```
app:
  icon: icon_64x64.png
```

Summary

Summary of the package used in Navigator.

```
app:
  summary:  "The Jupyter Notebook"
```

Own environment

If `True`, installing the app through Navigator installs into its own environment. The default is `False`.

```
app:
  own_environment: True
```

4.3.9 Extra section

A schema-free area for storing non-conda-specific metadata in standard YAML form.

EXAMPLE: To store recipe maintainer information:

```
extra:
  maintainers:
   - name of maintainer
```

4.3.10 Templating with Jinja

Conda-build supports Jinja templating in the `meta.yaml` file.

EXAMPLE: The following `meta.yaml` would work with the GIT values defined for Git repositores. The recipe is included at the base directory of the Git repository, so the `git_url` is `../`:

```
package:
  name: mypkg
  version: {{ GIT_DESCRIBE_TAG }}

build:
  number: {{ GIT_DESCRIBE_NUMBER }}

  # Note that this will override the default build string with the Python
  # and NumPy versions
  string: {{ GIT_BUILD_STR }}

source:
  git_url: ../
```

Conda-build checks if the Jinja2 variables that you use are defined and produces a clear error if it is not.

You can also use a different syntax for these environment variables that allows default values to be set, although it is somewhat more verbose.

EXAMPLE: A version of the previous example using the syntax that allows defaults:

```
package:
  name: mypkg
  version: {{ environ.get('GIT_DESCRIBE_TAG', '') }}

build:
  number: {{ environ.get('GIT_DESCRIBE_NUMBER', 0) }}

  # Note that this will override the default build string with the Python
  # and NumPy versions
  string: {{ environ.get('GIT_BUILD_STR', '') }}

source:
  git_url: ../
```

One further possibility using templating is obtaining data from your downloaded source code.

EXAMPLE: To process a project's `setup.py` and obtain the version and other metadata:

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

package:
  name: conda-build-test-source-setup-py-data
  version: {{ data.get('version') }}

# source will be downloaded prior to filling in jinja templates
# Example assumes that this folder has setup.py in it
source:
  path_url: ../
```

These functions are completely compatible with any other variables such as Git and Mercurial.

Extending this arbitrarily to other functions requires that functions be predefined before Jinja processing, which in practice means changing the conda-build source code. See the conda-build issue tracker.

For more information, see the Jinja2 template documentation and *the list of available environment variables*.

Jinja templates are evaluated during the build process. To retrieve a fully rendered `meta.yaml`, use the *conda render* command.

Conda-build specific Jinja2 functions

Besides the default Jinja2 functionality, additional Jinja functions are available during the conda-build process: `pin_compatible`, `pin_subpackage`, `compiler`, and `resolved_packages`. Please see *Extra Jinja2 functions* for the definition of the first 3 functions. Definition of `resolved_packages` is given below:

- `resolved_packages('environment_name')`: Returns the final list of packages (in the form of `package_name version build_string`) that are listed in `requirements:host` or `requirements:build`. This includes all packages (including the indirect dependencies) that will be installed in the host or build environment. `environment_name` must be either `host` or `build`. This function is useful for creating meta-packages that will want to pin all of their *direct* and *indirect* dependencies to their exact match. For example:

```
requirements:
  host:
    - curl 7.55.1
  run:
  {% for package in resolved_packages('host') %}
    - {{ package }}
  {% endfor %}
```

might render to (depending on package dependencies and the platform):

```
requirements:
    host:
        - ca-certificates 2017.08.26 h1d4fec5_0
        - curl 7.55.1 h78862de_4
        - libgcc-ng 7.2.0 h7cc24e2_2
        - libssh2 1.8.0 h9cfc8f7_4
        - openssl 1.0.2n hb7f436b_0
        - zlib 1.2.11 ha838bed_2
    run:
        - ca-certificates 2017.08.26 h1d4fec5_0
        - curl 7.55.1 h78862de_4
        - libgcc-ng 7.2.0 h7cc24e2_2
        - libssh2 1.8.0 h9cfc8f7_4
        - openssl 1.0.2n hb7f436b_0
        - zlib 1.2.11 ha838bed_2
```

Here, output of `resolved_packages` was:

```
['ca-certificates 2017.08.26 h1d4fec5_0', 'curl 7.55.1 h78862de_4',
'libgcc-ng 7.2.0 h7cc24e2_2', 'libssh2 1.8.0 h9cfc8f7_4',
'openssl 1.0.2n hb7f436b_0', 'zlib 1.2.11 ha838bed_2']
```

4.3.11 Preprocessing selectors

You can add selectors to any line, which are used as part of a preprocessing stage. Before the `meta.yaml` file is read, each selector is evaluated and if it is `False`, the line that it is on is removed. A selector has the form `# [<selector>]` at the end of a line.

```
source:
  url: http://path/to/unix/source    # [not win]
  url: http://path/to/windows/source # [win]
```

Note: Preprocessing selectors are evaluated after Jinja templates.

A selector is a valid Python statement that is executed. The following variables are defined. Unless otherwise stated, the variables are booleans.

x86	True if the system architecture is x86, both 32-bit and 64-bit, for Intel or AMD chips.
x86_64	True if the system architecture is x86_64, which is 64-bit, for Intel or AMD chips.
linux	True if the platform is Linux.
linux32	True if the platform is Linux and the Python architecture is 32-bit.
linux64	True if the platform is Linux and the Python architecture is 64-bit.
armv6l	True if the platform is Linux and the Python architecture is armv6l.
armv7l	True if the platform is Linux and the Python architecture is armv7l.
aarch64	True if the platform is Linux and the Python architecture is aarch64.
ppc64le	True if the platform is Linux and the Python architecture is ppc64le.
osx	True if the platform is macOS.
unix	True if the platform is either macOS or Linux.
win	True if the platform is Windows.
win32	True if the platform is Windows and the Python architecture is 32-bit.
win64	True if the platform is Windows and the Python architecture is 64-bit.
py	The Python version as an int, such as `27` or `36`. See the CONDA_PY *environment variable*.
py3k	True if the Python major version is 3.
py2k	True if the Python major version is 2.
py27	True if the Python version is 2.7. Use of this selector is discouraged in favor of comparison operators (e.g. py==27).
py34	True if the Python version is 3.4. Use of this selector is discouraged in favor of comparison operators (e.g. py==34).
py35	True if the Python version is 3.5. Use of this selector is discouraged in favor of comparison operators (e.g. py==35).
py36	True if the Python version is 3.6. Use of this selector is discouraged in favor of comparison operators (e.g. py==36).
np	The NumPy version as an integer such as `111`. See the CONDA_NPY *environment variable*.

The use of the Python version selectors, *py27*, *py34*, etc. is discouraged in favor of the more general comparison operators. Additional selectors in this series will not be added to conda-build.

Because the selector is any valid Python expression, complicated logic is possible:

```
source:
  url: http://path/to/windows/source      # [win]
  url: http://path/to/python2/unix/source # [unix and py2k]
  url: http://path/to/python3/unix/source # [unix and py>=35]
```

Note: The selectors delete only the line that they are on, so you may need to put the same selector on multiple lines:

```
source:
  url: http://path/to/windows/source     # [win]
  md5: 30fbf531409a18a48b1be249052e242a  # [win]
  url: http://path/to/unix/source        # [unix]
  md5: 88510902197cba0d1ab4791e0f41a66e  # [unix]
```

4.4 Adding pre-link, post-link, and pre-unlink scripts

You can add scripts to a recipe. They must be located in the same directory as the meta.yaml file. The following scripts can be added:

- `pre-link`---Executed before the package is installed. An error is indicated by a nonzero exit and causes conda to stop and causes the installation to fail.
- `post-link`---Executed after the package is installed. An error is indicated by a nonzero exist and causes installation to fail. If there is an error, conda does not write any package metadata.
- `pre-unlink`---Executed before the package is removed. An error is indicated by a nonzero exist and causes the removal to fail.

In addition to being co-located with the meta.yaml file, they must be named simply `post-link.sh` or `post-link.bat`. Conda-build will rename them to .<name>-<action>.sh (or .bat) where `<name>` is the package name and `<action>` is one of the preceeding actions.

These scripts are executed in a subprocess by conda, using `%COMSPEC% /c <script>` on Windows and `/bin/bash <script>` on macOS and Linux.

The convention for the path and filenames of these scripts on Windows is:

```
Scripts/.<name>-<action>.bat
```

On Linux and macOS the convention is:

```
bin/.<name>-<action>.sh
```

The scripts set the following environment variables:

PREFIX	The install prefix.
PKG_NAME	The name of the package.
PKG_VERSION	The version of the package.
PKG_BUILDNUM	The build number of the package.

The scripts are:

- Windows:
  - `pre-link.bat`
  - `post-link.bat`
  - `pre-unlink.bat`
- macOS and Linux:
  - `pre-link.sh`
  - `post-link.sh`
  - `pre-unlink.sh`

Post-link and pre-unlink scripts should:

- Be avoided whenever possible.
- Not touch anything other than the files being installed.
- Not write anything to stdout or stderr, unless an error occurs.
- Not depend on any installed or to-be-installed conda packages.

- Depend only on simple system tools such as `rm`, `cp`, `mv`, and `ln`.

The scripts should not write to stdout or stderr unless an error occurs, but they may write to `$PREFIX/.messages.txt`, which is shown after conda completes all actions.

4.5 Activate scripts

Recipes are allowed to have activate scripts which will be sourced or called when the environment is activated. It is generally recommended to avoid using activate scripts when another option is possible because people do not always activate environments the expected way and these packages may then misbehave.

When using them in a recipe, feel free to name them activate.bat, activate.sh, deactivate.bat, and deactivate.sh in the recipe. The installed scripts are recommended to be prefixed by the package name and a separating -.

Below is some sample code for Unix and Windows that will make this install process easier.

In `build.sh`:

```
# Copy the [de]activate scripts to $PREFIX/etc/conda/[de]activate.d.
# This will allow them to be run on environment activation.
for CHANGE in "activate" "deactivate"
do
    mkdir -p "${PREFIX}/etc/conda/${CHANGE}.d"
    cp "${RECIPE_DIR}/${CHANGE}.sh" "${PREFIX}/etc/conda/${CHANGE}.d/${PKG_NAME}_$
↪{CHANGE}.sh"
done
```

In `build.bat`:

```
setlocal EnableDelayedExpansion

:: Copy the [de]activate scripts to %PREFIX%\etc\conda\[de]activate.d.
:: This will allow them to be run on environment activation.
for %%F in (activate deactivate) DO (
    if not exist %PREFIX%\etc\conda\%%F.d mkdir %PREFIX%\etc\conda\%%F.d
    copy %RECIPE_DIR%\%%F.bat %PREFIX%\etc\conda\%%F.d\%PKG_NAME%_%%F.bat
    :: Copy unix shell activation scripts, needed by Windows Bash users
    copy %RECIPE_DIR%\%%F.sh %PREFIX%\etc\conda\%%F.d\%PKG_NAME%_%%F.sh
)
```

4.6 Making packages relocatable

Often, the most difficult thing about building a conda package is making it relocatable. Relocatable means that the package can be installed into any prefix. Otherwise, the package would be usable only in the environment in which it was built.

Conda-build does the following things automatically to make packages relocatable:

- Binary object files are converted to use relative paths using `install_name_tool` on macOS and patchelf on Linux.
- Any text file without NULL bytes that contains the build prefix or the placeholder prefix `/opt/anaconda1anaconda2anaconda3` is registered in the `info/has_prefix` file in the package metadata. When conda installs the package, any files in `info/has_prefix` have the registered prefix replaced with the install prefix. For more information, see *Package metadata*.

- Any binary file containing the build prefix can automatically be registered in `info/has_prefix` using `build/detect_binary_files_with_prefix` in `meta.yaml`. Alternatively, individual binary files can be registered by listing them in `build/binary_has_prefix_files` in `meta.yaml`. The registered files will have their build prefix replaced with the install prefix at install time. This works by padding the install prefix with null terminators, such that the length of the binary file remains the same. The build prefix must therefore be long enough to accommodate any reasonable installation prefix. On macOS and Linux, conda-build pads the build prefix to 255 characters by appending `_placehold`s to the end of the build directory name.

Note: The prefix length was changed in conda-build 2.0 from 80 characters to 255 characters. Legacy packages with 80-character prefixes must be rebuilt to take advantage of the longer prefix.

- There may be cases where conda identified a file as binary, but it needs to have the build prefix replaced as if it were text---no padding with null terminators. Such files can be listed in `build/has_prefix_files` in `meta.yaml`.

4.7 Conda package specification

- *Package metadata*
- *Link and unlink scripts*
- *Repository structure and index*
- *Package match specifications*

A conda package is a bzipped tar archive---.tar.bz2---that contains:

- Metadata under the `info/` directory.
- A collection of files that are installed directly into an install prefix.

The format is identical across platforms and operating systems. During the install process, all files are extracted into the install prefix, with the exception of the ones in `info/`. Installing a conda package into an environment is similar to executing the following commands:

```
cd <environment prefix>
tar xjf some-package-1.0-0.tar.bz2
```

Only files, including symbolic links, are part of a conda package. Directories are not included. Directories are created and removed as needed, but you cannot create an empty directory from the tar archive directly.

4.7.1 Package metadata

The `info/` directory contains all metadata about a package. Files in this location are not installed under the install prefix. Although you are free to add any file to this directory, conda only inspects the content of the files discussed below.

info/index.json

This file contains basic information about the package, such as name, version, build string, and dependencies. The content of this file is stored in `repodata.json`, which is the repository index file, hence the name `index.json`. The JSON object is a dictionary containing the keys shown below. The filename of the conda package is composed of the first 3 values, as in: `<name>-<version>-<build>.tar.bz2`.

Key	Type	Description
name	string	The lowercase name of the package. May contain the "-" character.
version	string	The package version. May not contain "-". Conda acknowledges PEP 440.
build	string	The build string. May not contain "-". Differentiates builds of packages with otherwise identical names and versions, such as: • A build with other dependencies, such as Python 3.4 instead of Python 2.7. • A bug fix in the build process. • Some different optional dependencies, such as MKL versus ATLAS linkage. Nothing in conda actually inspects the build string. Strings such as `np18py34_1` are designed only for human readability and conda never parses them.
build_number	integer	A non-negative integer representing the build number of the package. Unlike the build string, the `build_number` is inspected by conda. Conda uses it to sort packages that have otherwise identical names and versions to determine the latest one. This is important because new builds that contain bug fixes for the way a package is built may be added to a repository.
depends	list of strings	A list of dependency specifications, where each element is a string, as outlined in *Package match specifications*.
arch	string	Optional. The architecture the package is built for. EXAMPLE: `x86_64` Conda currently does not use this key.
platform	string	Optional. The OS that the package is built for. EXAMPLE: `osx` Conda currently does not use this key. Packages for a specific architecture and platform are usually distinguished by the repository subdirectory that contains them---see *Repository structure and index*.

info/files

Lists all files that are part of the package itself, 1 per line. All of these files need to get linked into the environment. Any files in the package that are not listed in this file are not linked when the package is installed. The directory delimiter for the files in `info/files` should always be "/", even on Windows. This matches the directory delimiter used in the tarball.

info/has_prefix

Optional file. Lists all files that contain a hard-coded build prefix or placeholder prefix, which needs to be replaced by the install prefix at installation time.

Note: Due to the way the binary replacement works, the placeholder prefix must be longer than the install prefix.

Each line of this file should be either a path, in which case it is considered a text file with the default placeholder `/opt/anaconda1anaconda2anaconda3`, or a space-separated list of placeholder, mode, and path, where:

- Placeholder is the build or placeholder prefix.
- Mode is either `text` or `binary`.
- Path is the relative path of the file to be updated.

EXAMPLE: On Windows:

```
"Scripts/script1.py"
"C:\Users\username\anaconda\envs\_build" text "Scripts/script2.bat"
"C:/Users/username/anaconda/envs/_build" binary "Scripts/binary"
```

EXAMPLE: On macOS or Linux:

```
bin/script.sh
/Users/username/anaconda/envs/_build binary bin/binary
/Users/username/anaconda/envs/_build text share/text
```

Note: The directory delimiter for the relative path must always be "/", even on Windows. The placeholder may contain either "\" or "/" on Windows, but the replacement prefix will match the delimiter used in the placeholder. The default placeholder `/opt/anaconda1anaconda2anaconda3` is an exception, being replaced with the install prefix using the native path delimiter. On Windows, the placeholder and path always appear in quotes to support paths with spaces.

info/license.txt

Optional file. The software license for the package.

info/no_link

Optional file. Lists all files that cannot be linked---either soft-linked or hard-linked---into environments and are copied instead.

info/about.json

Optional file. Contains the entries in the *About section* of the `meta.yaml` file. The following keys are added to `info/about.json` if present in the build recipe:

- home.
- dev_url.
- doc_url.
- license_url.
- license.
- summary.
- description.
- license_family.

info/recipe

A directory containing the full contents of the build recipe.

meta.yaml.rendered

The fully rendered build recipe. See *conda render*.

This directory is present only when the the `include_recipe` flag is `True` in the *Build section*.

4.7.2 Link and unlink scripts

You may optionally execute scripts before and after the link and unlink steps. For more information, see *Adding pre-link, post-link, and pre-unlink scripts*.

4.7.3 Repository structure and index

A conda repository---or channel---is a directory tree, usually served over HTTPS, which has platform subdirectories, each of which contains conda packages and a repository index. The index file `repodata.json` lists all conda packages in the platform subdirectory. Use `conda index` to create such an index from the conda packages within a directory. It is simple mapping of the full conda package filename to the dictionary object in `info/index.json` described in *Adding pre-link, post-link, and pre-unlink scripts*.

In the following example, a repository provides the conda package `misc-1.0-np17py27_0.tar.bz2` on 64-bit Linux and 32-bit Windows:

```
<some path>/linux-64/repodata.json
                     repodata.json.bz2
                     misc-1.0-np17py27_0.tar.bz2
           /win-32/repodata.json
                   repodata.json.bz2
                   misc-1.0-np17py27_0.tar.bz2
```

Note: Both conda packages have identical filenames and are distinguished only by the repository subdirectory that contains them.

4.7.4 Package match specifications

This match specification is not the same as the syntax used at the command line with `conda install`, such as `conda install python=3.4`. Internally, conda translates the command line syntax to the spec defined in this section.

EXAMPLE: python=3.4 is translated to python 3.4*.

Package dependencies are specified using a match specification. A match specification is a space-separated string of 1, 2, or 3 parts:

- The first part is always the exact name of the package.
- The second part refers to the version and may contain special characters:
  - | means OR.

    EXAMPLE: `1.0|1.2` matches version 1.0 or 1.2.
  - * matches 0 or more characters in the version string. In terms of regular expressions, it is the same as `r'.*'`.

    EXAMPLE: 1.0|1.4* matches 1.0, 1.4 and 1.4.1b2, but not 1.2.
  - <, >, <=, >=, ==, and != are relational operators on versions, which are compared using PEP-440. For example, `<=1.0` matches `0.9`, `0.9.1`, and `1.0`, but not `1.0.1`. `==` and `!=` are exact equality.

    Pre-release versioning is also supported such that `>1.0b4` will match `1.0b5` and `1.0rc1` but not `1.0b4` or `1.0a5`.

    EXAMPLE: <=1.0 matches 0.9, 0.9.1, and 1.0, but not 1.0.1.
  - , means AND.

    EXAMPLE: >=2,<3 matches all packages in the 2 series. 2.0, 2.1, and 2.9 all match, but 3.0 and 1.0 do not.
  - , has higher precedence than |, so >=1,<2|>3 means greater than or equal to 1 AND less than 2 or greater than 3, which matches 1, 1.3 and 3.0, but not 2.2.

  Conda parses the version by splitting it into parts separated by |. If the part begins with <, >, =, or !, it is parsed as a relational operator. Otherwise, it is parsed as a version, possibly containing the "*" operator.
- The third part is always the exact build string. When there are 3 parts, the second part must be the exact version.

Remember that the version specification cannot contain spaces, as spaces are used to delimit the package, version, and build string in the whole match specification. `python >= 2.7` is an invalid match specification. Furthermore, `python>=2.7` is matched as any version of a package named `python>=2.7`.

When using the command line, put double quotes around any package version specification that contains the space character or any of the following characters: <, >, *, or |.

EXAMPLE:

```
conda install numpy=1.11
conda install numpy==1.11
conda install "numpy>1.11"
```

(continues on next page)

(continued from previous page)

```
conda install "numpy=1.11.1|1.11.3"
conda install "numpy>=1.8,<2"
```

Examples

The OR constraint "numpy=1.11.1|1.11.3" matches with 1.11.1 or 1.11.3.

The AND constraint "numpy>=1.8,<2" matches with 1.8 and 1.9 but not 2.0.

The fuzzy constraint numpy=1.11 matches 1.11, 1.11.0, 1.11.1, 1.11.2, 1.11.18, and so on.

The exact constraint numpy==1.11 matches 1.11, 1.11.0, 1.11.0.0, and so on.

The build string constraint "numpy=1.11.2=*nomkl*" matches the NumPy 1.11.2 packages without MKL but not the normal MKL NumPy 1.11.2 packages.

The build string constraint "numpy=1.11.1|1.11.3=py36_0" matches NumPy 1.11.1 or 1.11.3 built for Python 3.6 but not any versions of NumPy built for Python 3.5 or Python 2.7.

The following are all valid match specifications for numpy-1.8.1-py27_0:

- numpy
- numpy 1.8*
- numpy 1.8.1
- numpy >=1.8
- numpy ==1.8.1
- numpy 1.8|1.8*
- numpy >=1.8,<2
- numpy >=1.8,<2|1.9
- numpy 1.8.1 py27_0
- numpy=1.8.1=py27_0

4.8 Using shared libraries

Shared libraries are libraries that are loosely coupled to the programs and extensions that depend on them. When loading an executable into memory, an operating system finds all dependent shared libraries and links them to the executable so that it can run.

Windows, macOS, and Linux all provide a way to build executables and libraries that contain links to the shared libraries they depend on, instead of directly linking the libraries themselves.

4.8.1 Shared libraries in Windows

Unlike macOS and Linux, Windows does not have the concept of embedding links into binaries. Instead, Windows depends primarily on searching directories for matching filenames, as documented in Search Path Used by Windows to Locate a DLL.

There is an alternate configuration, called side-by-side assemblies, that requires specification of DLL versions in either an embedded manifest or an appropriately named XML file alongside the binary in question. Conda does not currently use side-by-side assemblies, but it may turn towards that in the future to resolve complications with multiple versions of the same library on the same system.

For now, most DLLs are installed into `(install prefix)\\Library\\bin`. This path is added to `os.environ["PATH"]` for all Python processes, so that DLLs can be located, regardless of the value of the system's PATH environment variable.

Note: PATH is searched from left to right, with the first DLL name match being picked up, in the absence of a manifest specifying otherwise. This means that installing software with other matching DLLs may give you a system that crashes in unpredictable ways. When troubleshooting or asking for support on Windows, always consider PATH as a potential source of issues.

4.8.2 Shared libraries in macOS and Linux

In macOS and Linux, dynamic links are discovered in a similar manner to the way that Python modules are discovered via PYTHONPATH, and executables are discovered via PATH. A list of search locations is made, and then the library objects are searched for in the search locations. By default, as well as by design, the system dynamic linker does not have any special preference for the conda environment `lib` directories.

You can specify both absolute links and relative links. If the links are absolute paths, such as `/Users/UserName/my_build_env`, the library works only on a system where that exact path exists. Therefore, relative links are preferred in conda packages.

Relative links require a special variable in the link itself:

- On Linux, the $ORIGIN variable allows you to specify "relative to this file as it is being executed".
- On macOS, the variables are:
  - @rpath---Allows you to set relative links from the system load paths.
  - @loader_path---Equivalent to $ORIGIN.
  - @executable_path---Supports the Apple `.app` directory approach, where libraries know where they live relative to their calling application.

Conda-build uses @loader_path on macOS and $ORIGIN on Linux because we install into a common root directory and can assume that other libraries are also installed into that root. The use of the variables allows you to build relocatable binaries that can be built on one system and sent everywhere.

On Linux, `conda-build` modifies any shared libraries or generated executables to use a relative dynamic link by calling the patchelf tool. On macOS, the install_name_tool tool is used.

Warning: Setting LD_LIBRARY_PATH on Linux or DYLD_LIBRARY_PATH on macOS can interfere with this because the dynamic linker short-circuits link resolution by first looking at LD_LIBRARY_PATH.

EXAMPLE: You install an old version of libcurl into your conda environment due to some compatibility issues with the code you're using. Then, you set `export LD_LIBRARY_PATH=/home/UserName/envs/curl_env/lib`. From that point on, every program that you execute in that session will favor this libcurl to your system libcurl because it is now effectively at the "front" of the dynamic load path.

Including conda environment paths in LD_LIBRARY_PATH or DYLD_LIBRARY_PATH is not recommended.

4.9 Build variants

The nature of binary compatibility (and incompatibility) means that we sometimes need to build binary packages (and any package containing binaries) with several variants to support different usage environments. For example, using NumPy's C API means that a package must be used with the same version of NumPy at runtime that was used at build time.

There has been limited support for this for a long time. Including Python in both build and run requirements resulted in a package with Python pinned to the version of Python used at build time, and a corresponding addition to the filename such as "py27". Similar support existed for NumPy with the addition of an `x.x` pin in the recipe after Conda-build PR 573 was merged. Before conda-build version 3.0, there were also many longstanding proposals for general support (Conda-build issue 1142).

As of conda-build 3.0, a new configuration scheme has been added, dubbed "variants." Conceptually, this decouples pinning values from recipes, replacing them with Jinja2 template variables. It adds support for the notion of "compatible" pinnings to be integrated with ABI compatibility databases, such as ABI Laboratory. Note that the concept of "compatible" pinnings is currently still under heavy development.

Variant input is ultimately a dictionary. These dictionaries are mostly very flat. Keys are made directly available in Jinja2 templates. As a result, keys in the dictionary (and in files read into dictionaries) must be valid jinja2 variable names (no `-` characters allowed). This example builds Python 2.7 and 3.5 packages in one build command:

conda_build_config.yaml like:

```yaml
python:
    - 2.7
    - 3.5
```

meta.yaml contents like:

```yaml
package:
    name: compiled-code
    version: 1.0

requirements:
    build:
        - python {{ python }}
    run:
        - python
```

The command to build recipes is unchanged relative to earlier conda-build versions. For example, with our shell in the same folder as meta.yaml and conda_build_config.yaml, we just call the `conda build .` command.

4.9.1 General pinning examples

There are a few characteristic use cases for pinning. Please consider this a map for the content below.

1. Shared library providing a binary interface. All uses of this library use the binary interface. It is convenient to apply the same pin to all of your builds. Example: boost

   conda_build_config.yaml in your HOME folder:

```
boost:
  - 1.61
  - 1.63
pin_run_as_build:
  boost: x.x
```

   meta.yaml:

```
package:
    name: compiled-code
    version: 1.0

requirements:
    build:
        - boost  {{ boost }}
    run:
        - boost
```

   This example demonstrates several features:

   - User-wide configuration with a specifically named config file (conda_build_config.yaml in your home folder). More options below in *Creating conda-build variant config files*.
   - Building against multiple versions of a single library (set versions installed at build time).
   - Pinning runtime requirements to the version used at build time. More information below at *Pinning at the variant level*.
   - Specify granularity of pinning. `x.x` pins major and minor version. More information at *Pinning expressions*.

2. Python package with externally accessible binary component. Not all uses of this library use the binary interface (some only use pure Python). Example: NumPy.

   conda_build_config.yaml in your recipe folder (alongside meta.yaml):

```
numpy:
  - 1.11
  - 1.12
```

   meta.yaml:

```
package:
    name: numpy_using_pythonAPI_thing
    version: 1.0

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

(continues on next page)

(continued from previous page)

```
        - python
        - numpy
```

This example demonstrates a particular feature: reduction of builds when pins are unnecessary. Since the example recipe above only requires the Python API to NumPy, we will only build the package once and the version of NumPy will not be pinned at runtime to match the compile-time version. There's more information at *Avoiding unnecessary builds*.

For a different package that makes use of the NumPy C API, we will need to actually pin NumPy in this recipe (and only in this recipe, so that other recipes don't unnecessarily build lots of variants). To pin NumPy, you can use the variant key directly in meta.yaml:

```
package:
    name: numpy_using_cAPI_thing
    version: 1.0

requirements:
    build:
        - numpy  {{ numpy }}
    run:
        - numpy  {{ numpy }}
```

For legacy compatibility, Python is pinned implicitly without specifying `{{ python }}` in your recipe. This is generally intractable to extend to all package names, so in general, try to get in the habit of always using the Jinja2 variable substitution for pinning using versions from your conda_build_config.yaml file.

There are also more flexible ways to pin using the *Pinning expressions*. See *Pinning at the recipe level* for examples.

3. One recipe splits into multiple packages, and package dependencies need to be dynamically pinned among one another. Example: GCC/libgcc/libstdc++/gfortran/etc.

   The dynamic pinning is the tricky part. Conda-build provides new ways to refer to other subpackages within a single recipe.

```
package:
    name: dynamic_supackage
    version: 1.0

requirements:
    run:
        - {{ pin_subpackage('my_awesome_subpackage') }}

outputs:
  - name: my_awesome_subpackage
    version: 2.0
```

   By referring to subpackages this way, you don't need to worry about what the end version of `my_awesome_subpackage` will be. Update it independently and just let conda-build figure it out and keep things consistent. There's more information below in the *Referencing subpackages* section.

4.9.2 Transition guide

Let's say we have a set of recipes that currently builds a C library, as well as Python and R bindings to that C library. xgboost, a recent machine learning library, is one such example. Under conda-build 2.0 and earlier, you needed to have 3 recipes - 1 for each component. Let's go over some simplified `meta.yaml` files. First, the C library:

```
package:
    name: libxgboost
    version: 1.0
```

Next, the Python bindings:

```
package:
    name: py-xgboost
    version: 1.0

requirements:
    build:
        - libxgboost  # you probably want to pin the version here, but there's no␣
↪dynamic way to do it
        - python
    run:
        - libxgboost  # you probably want to pin the version here, but there's no␣
↪dynamic way to do it
        - python
```

```
package:
    name: r-xgboost
    version: 1.0

requirements:
    build:
        - libxgboost  # you probably want to pin the version here, but there's no␣
↪dynamic way to do it
        - r-base
    run:
        - libxgboost  # you probably want to pin the version here, but there's no␣
↪dynamic way to do it
        - r-base
```

To build these, you'd need several conda-build commands, or a tool like conda-build-all to build out the various Python versions. With conda-build 3.0 and split packages from conda-build 2.1, we can simplify this to one coherent recipe that also includes the matrix of all desired Python and R builds.

First, the `meta.yaml` file:

```
package:
    name: xgboost
    version: 1.0

outputs:
    - name: libxgboost
    - name: py-xgboost
      requirements:
          - {{ pin_subpackage('libxgboost', exact=True) }}
          - python  {{ python }}

```

(continues on next page)

(continued from previous page)

```
    - name: r-xgboost
      requirements:
          - {{ pin_subpackage('libxgboost', exact=True) }}
          - r-base  {{ r_base }}
```

Next, the `conda_build_config.yaml` file, specifying our build matrix:

```
python:
    - 2.7
    - 3.5
    - 3.6
r_base:
    - 3.3.2
    - 3.4.0
```

With this updated method, you get a complete build matrix: 6 builds total. One libxgboost library, 3 Python versions, and 2 R versions. Additionally, the Python and R packages will have exact pins to the libxgboost package that was built by this recipe.

4.9.3 Creating conda-build variant config files

Variant input files are yaml files. Search order for these files is the following:

1. A file named `conda_build_config.yaml` in the user's HOME folder (or an arbitrarily named file specified as the value for the `conda_build/config_file` key in your .condarc file).
2. A file named `conda_build_config.yaml` in the current working directory.
3. A file named `conda_build_config.yaml` in the same folder as `meta.yaml` with your recipe.
4. Any additional files specified on the command line with the `--variant-config-files` or `-m` command line flags, which can be passed multiple times for multiple files. The `conda build` and `conda render` commands accept these arguments.

Values in files found later in this search order will overwrite and replace the values from earlier files.

Note: The key `conda_build/config_file` is a nested value:

```
conda_build:
  config_file: some/path/to/file
```

4.9.4 Using variants with the conda-build API

Ultimately, a variant is just a dictionary. This dictionary is provided directly to Jinja2 and you can use any declared key from your variant configuration in your Jinja2 templates. There are two ways that you can feed this information into the API:

1. Pass the `variants` keyword argument to API functions. Currently, the `build`, `render`, `get_output_file_path`, and `check` functions accept this argument. `variants` should be a dictionary where each value is a list of versions to iterate over. These are aggregated as detailed in the *Aggregation of multiple variants* section below.
2. Set the `variant` member of a Config object. This is just a dictionary. The values for fields should be strings or lists of strings, except "extended keys", which are documented in the *Extended keys* section below.

Again, with `meta.yaml` contents like:

```
package:
    name: compiled-code
    version: 1.0

requirements:
    build:
        - python {{ python }}
    run:
        - python {{ python }}
```

You could supply a variant to build this recipe like so:

```
variants = {'python': ['2.7', '3.5']}
api.build(path_to_recipe, variants=variants)
```

Note that these Jinja2 variable substitutions are not limited to version numbers. You can use them anywhere, for any string value. For example, to build against different MPI implementations:

With `meta.yaml` contents like:

```
package:
    name: compiled-code
    version: 1.0

requirements:
    build:
        - {{ mpi }}
    run:
        - {{ mpi }}
```

You could supply a variant to build this recipe like this (`conda_build_config.yaml`):

```
mpi:
    - openmpi  # version spec here is totally valid, and will apply in the recipe
    - mpich  # version spec here is totally valid, and will apply in the recipe
```

Selectors are valid in `conda_build_config.yaml`, so you can have one `conda_build_config.yaml` for multiple platforms:

```
mpi:
    - openmpi  # [osx]
    - mpich    # [linux]
    - msmpi    # [win]
```

Jinja is not allowed in `conda_build_config.yaml`, though. It is the source of information to feed into other Jinja templates, and the buck has to stop somewhere.

4.9.5 About reproducibility

A critical part of any build system is ensuring that you can reproduce the same output at some future point in time. This is often essential for troubleshooting bugs. For example, if a package contains only binaries, it is helpful to understand what source code created those binaries, and thus what bugs might be present.

Since conda-build 2.0, conda-build has recorded its rendered `meta.yaml` files into the `info/recipe` folder of each package it builds. Conda-build 3.0 is no different in this regard, but the `meta.yaml` that is recorded is a frozen set of the variables that make up the variant for that build.

Note: Package builders may disable including the recipe with the `build/include_recipe` key in `meta.yaml`. If the recipe is omitted from the package, then the package is not reproducible without the source recipe.

4.9.6 Special variant keys

There are some special keys that behave differently and can be more nested:

- `zip_keys`: a list of strings or a list of lists of strings. Strings are keys in variant. These couple groups of keys, so that particular keys are paired, rather than forming a matrix. This is useful, for example, to couple vc version to Python version on Windows. More info below in the *Coupling keys* section.
- `pin_run_as_build`: should be a dictionary. Keys are package names. Values are "pinning expressions" - explained in more detail in *Customizing compatibility*. This is a generalization of the `numpy x.x` spec, so that you can pin your packages dynamically based on the versions used at build time.
- `extend_keys`: specifies keys that should be aggregated, and not replaced, by later variants. These are detailed below in the *Extended keys* section.
- `ignore_version`: list of package names whose versions should be excluded from `meta.yaml`'s requirements/build when computing hash. Described further in *Avoiding unnecessary builds*.

4.9.7 Coupling keys

Sometimes particular versions need to be tied to other versions. For example, on Windows, we generally follow the upstream Python.org association of Visual Studio compiler version with Python version. Python 2.7 is always compiled with Visual Studio 2008 (also known as MSVC 9). We don't want a `conda_build_config.yaml` like the following to create a matrix of Python/MSVC versions:

```yaml
python:
  - 2.7
  - 3.5
vc:
  - 9
  - 14
```

Instead, we want 2.7 to be associated with 9, and 3.5 to be associated with 14. The `zip_keys` key in `conda_build_config.yaml` is the way to achieve this:

```yaml
python:
  - 2.7
  - 3.5
vc:
  - 9
  - 14
```

(continues on next page)

(continued from previous page)

```
zip_keys:
  - python
  - vc
```

You can also have nested lists to achieve multiple groups of `zip_keys`:

```
zip_keys:
  -
    - python
    - vc
  -
    - numpy
    - blas
```

The rules for `zip_keys` are:

1. Every list in a group must be the same length. This is because without equal length, there is no way to associate earlier elements from the shorter list with later elements in the longer list. For example, this is invalid, and will raise an error:

   ```
   python:
     - 2.7
     - 3.5
   vc:
     - 9
   zip_keys:
     - python
     - vc
   ```

2. `zip_keys` must be either a list of strings, or a list of lists of strings. You can't mix them. For example, this is an error:

   ```
   zip_keys:
     -
       - python
       - vc
     - numpy
     - blas
   ```

Rule #1 raises an interesting use case: How does one combine CLI flags like --python with `zip_keys`? Such a CLI flag will change the variant so that it has only a single entry, but it will not change the `vc` entry in the variant configuration. We'll end up with mismatched list lengths, and an error. To overcome this, you should instead write a very simple YAML file with all involved keys. Let's call it `python27.yaml`, to reflect its intent:

```
python:
  - 2.7
vc:
  - 9
```

Provide this file as a command-line argument:

```
conda build recipe -m python27.yaml
```

You can also specify variants in JSON notation from the CLI as detailed in the *CONDA_* variables and command line arguments to conda-build* section. For example:

```
conda build recipe --variants "{'python': ['2.7', '3.5'], 'vc': ['9', '14']}"
```

4.9.8 Avoiding unnecessary builds

To avoid building variants of packages where pinning does not require having different builds, you can use the `ignore_version` key in your variant. Then all variants are evaluated, but if any hashes are the same, then they are considered duplicates, and are deduplicated. By omitting some packages from the build dependencies, we can avoid creating unnecessarily specific hashes and allow this deduplication.

For example, let's consider a package that uses NumPy in both run and build requirements, and a variant that includes 2 NumPy versions:

```
variants = [{'numpy': ['1.10', '1.11'], 'ignore_version': ['numpy']}]
```

`meta.yaml`:

```
requirements:
    build:
        - numpy {{ numpy }}
    run:
        - numpy
```

Here, the variant says that we'll have two builds - one for each NumPy version. However, since this recipe does not pin NumPy's run requirement (because it doesn't utilize NumPy's C API), it is unnecessary to build it against both NumPy 1.10 and 1.11.

The rendered form of this recipe, with conda-build ignoring NumPy's value in the recipe, is going to be just one build that looks like:

`meta.yaml`:

```
requirements:
    build:
        - numpy
    run:
        - numpy
```

`ignore_version` is an empty list by default. The actual build performed is probably done with the last 'numpy' list element in the variant, but that's an implementation detail that you should not depend on. The order is considered unspecified behavior because the output should be independent of the input versions.

Warning: If the output is not independent of input versions, don't use this key

Any pinning done in the run requirements will affect the hash, and thus builds will be done for each variant in the matrix. Any package that sometimes is used for its compiled interface and sometimes used for only its Python interface may benefit from careful use of `ignore_version` in the latter case.

Note: `pin_run_as_build` is kind of the opposite of `ignore_version`. Where they conflict, `pin_run_as_build` takes priority.

4.9.9 CONDA_* variables and command line arguments to conda-build

To ensure consistency with existing users of conda-build, environment variables such as CONDA_PY behave as they always have, and they overwrite all variants set in files or passed to the API.

The full list of respected environment variables are:

- CONDA_PY
- CONDA_NPY
- CONDA_R
- CONDA_PERL
- CONDA_LUA

CLI flags are also still available. These are sticking around for their usefulness in one-off jobs.

- --python
- --numpy
- --R
- --perl
- --lua

In addition to these traditional options, there's one new flag to specify variants: `--variants`. This flag accepts a string of JSON-formatted text. For example:

```
conda build recipe --variants "{python: [2.7, 3.5], vc: [9, 14]}"
```

4.9.10 Aggregation of multiple variants

The matrix of all variants is first consolidated from several dicts of lists into a single dict of lists, and then transformed in a list of dicts (using the Cartesian product of lists), where each value is a single string from the list of potential values.

For example, general input for `variants` could be something like:

```
a = {'python': ['2.7', '3.5'], 'numpy': ['1.10', '1.11']}
# values can be strings or lists.  Strings are converted to one-element lists␣
↪internally.
b = {'python': ['3.4', '3.5'], 'numpy': '1.11'}
```

Here, let's say `b` is found after `a`, and thus has priority over `a`. Merging these 2 variants yields:

```
merged = {'python': ['3.4', '3.5'], 'numpy': ['1.11']}
```

`b`'s values for `python` have overwritten `a`'s. From here, we compute the Cartesian product of all input variables. The end result is a collection of dicts, each with a string for each value. Output would be something like:

```
variants = [{'python': '3.4', 'numpy': '1.11'}, {'python': '3.5', 'numpy': '1.11'}]
```

conda-build would loop over these variants where appropriate, such as when building, outputting package output names, and so on.

If `numpy` had had two values instead of one, we'd end up with *four* output variants: 2 variants for `python`, *times* 2 variants for `numpy`:

```
variants = [{'python': '3.4', 'numpy': '1.11'}, {'python': '3.5', 'numpy': '1.11'},
            {'python': '3.4', 'numpy': '1.10'}, {'python': '3.5', 'numpy': '1.10'}]
```

4.9.11 Bootstrapping pins based on an existing environment

To establish your initial variant, you may point to an existing conda environment. Conda-build will examine the contents of that environment and pin to the exact requirements that make up that environment.

```
conda build --bootstrap name_of_env
```

You may specify either environment name or filesystem path to the environment. Note that specifying environment name does mean depending on conda's environment lookup.

4.9.12 Extended keys

These are not looped over to establish the build matrix. Rather, they are aggregated from all input variants, and each derived variant shares the whole set. These are used internally for tracking which requirements should be pinned, for example, with the `pin_run_as_build` key. You can add your own extended keys by passing in values for the `extend_keys` key for any variant.

For example, if you wanted to collect some aggregate trait from multiple `conda_build_config.yaml` files, you could do something like this:

`HOME/conda_build_config.yaml`:

```
some_trait:
  - dog
extend_keys:
  - some_trait
```

`recipe/conda_build_config.yaml`:

```
some_trait:
  - pony
extend_keys:
  - some_trait
```

Note that *both* of the `conda_build_config.yaml` files need to list the trait as an `extend_keys` entry. If you list it in only one of them, an error will be raised to avoid confusion with one `conda_build_config.yaml` file that would add entries to the build matrix, and another which would not. For example, this should raise an error:

```
some_trait:
  - dog
```

`recipe/conda_build_config.yaml`:

```
some_trait:
  - pony
extend_keys:
  - some_trait
```

When our two proper YAML config files are combined, ordinarily the recipe-local variant would overwrite the user-wide variant, yielding `{'some_trait': 'pony'}`. However, with the `extend_keys` entry, we end up with what we've always wanted: a dog *and* pony show: `{'some_trait': ['dog', 'pony'])}`

Again, this is mostly an internal implementation detail - unless you find a use for it. Internally, it is used to aggregate the `pin_run_as_build` and `ignore_version` entries from any of your `conda_build_config.yaml` files.

4.9.13 Customizing compatibility

Pinning expressions

Pinning expressions are the syntax used to specify how many parts of the version to pin. They are by convention strings containing `x` characters separated by `.`. The number of version parts to pin is simply the number of things that are separated by `.`. For example, `"x.x"` pins major and minor version. `"x"` pins only major version.

Wherever pinning expressions are accepted, you can customize both lower and upper bounds.

```
# produces pins like >=1.11.2,<1.12
variants = [{'numpy': '1.11', 'pin_run_as_build': {'numpy': {'max_pin': 'x.x'}}}]
```

Note that the final pin may be more specific than your initial spec. Here, the spec is 1.11, but the produced pin could be 1.11.2, the exact version of NumPy that was used at build time.

```
# produces pins like >=1.11,<2
variants = [{'numpy': '1.11', 'pin_run_as_build': {'numpy': {'min_pin': 'x.x', 'max_
↪pin': 'x'}}}]
```

Pinning at the variant level

Some packages, such as boost, *always* need to be pinned at runtime to the version that was present at build time. For these cases where the need for pinning is consistent, pinning at the variant level is a good option. Conda-build will automatically pin run requirements to the versions present in the build environment when the following conditions are met:

1. The dependency is listed in the requirements/build section. It can be pinned, but does not need to be.
2. The dependency is listed by name (no pinning) in the requirements/run section.
3. The `pin_run_as_build` key in the variant has a value that is a dictionary, containing a key that matches the dependency name listed in the run requirements. The value should be a dictionary with up to 4 keys: `min_pin`, `max_pin`, `lower_bound`, `upper_bound`. The first 2 are pinning expressions. The latter 2 are version numbers, overriding detection of current version.

An example variant/recipe is shown here:

`conda_build_config.yaml`:

```
boost: 1.63
pin_run_as_build:
    boost:
      max_pin: x.x
```

`meta.yaml`:

```
requirements:
    build:
        - boost {{ boost }}
    run:
        - boost
```

The result here is that the runtime boost dependency will be pinned to `>=(current boost 1.63.x version),<1.64`.

More details on the `pin_run_as_build` function is below in the *Extra Jinja2 functions* section.

Note that there are some packages that you should not use `pin_run_as_build` for. Packages that don't *always* need to be pinned should be pinned on a per-recipe basis (described in the next section). NumPy is an interesting example here. It actually would not make a good case for pinning at the variant level. Because you only need this kind of pinning for recipes that use NumPy's C API, it would actually be better not to pin NumPy with `pin_run_as_build`. Pinning it is over-constraining your requirements unnecessarily when you are not using NumPy's C API. Instead, we should customize it for each recipe that uses NumPy. See also the *Avoiding unnecessary builds* section above.

Pinning at the recipe level

Pinning at the recipe level overrides pinning at the variant level, because run dependencies that have pinning values in `meta.yaml` (even as Jinja variables) are ignored by the logic handling `pin_run_as_build`. We expect that pinning at the recipe level will be used when some recipe's pinning is unusually stringent (or loose) relative to some standard pinning from the variant level.

By default, with the `pin_compatible('package_name')` function, conda-build pins to your current version and less than the next major version. For projects that don't follow the philosophy of semantic versioning, you might want to restrict things more tightly. To do so, you can pass one of two arguments to the `pin_compatible` function.

```
variants = [{'numpy': '1.11'}]
```

`meta.yaml`:

```
requirements:
    build:
        - numpy {{ numpy }}
    run:
        - {{ pin_compatible('numpy', max_pin='x.x') }}
```

This would yield a pinning of `>=1.11.2,<1.12`.

The syntax for the `min_pin` and `max_pin` is a string pinning expression. Each can be passed independently of the other. An example of specifying both:

```
variants = [{'numpy': '1.11'}]
```

`meta.yaml`:

```
requirements:
    build:
        - numpy {{ numpy }}
    run:
        - {{ pin_compatible('numpy', min_pin='x.x', max_pin='x.x') }}
```

This would yield a pinning of `>=1.11,<1.12`.

You can also pass the minimum or maximum version directly. These arguments supersede the `min_pin` and `max_pin` arguments and are thus mutually exclusive.

```
variants = [{'numpy': '1.11'}]
```

`meta.yaml`:

```
requirements:
    build:
        - numpy {{ numpy }}
    run:
        - {{ pin_compatible('numpy', lower_bound='1.10', upper_bound='3.0') }}
```

This would yield a pinning of `>=1.10,<3.0`.

4.9.14 Appending to recipes

As of conda-build 3.0, you can add a file named `recipe_append.yaml` in the same folder as your `meta.yaml` file. This file is considered to follow the same rules as `meta.yaml`, except that selectors and Jinja2 templates are not evaluated. Evaluation of selectors and Jinja2 templates will likely be added in future development.

Any contents in `recipe_append.yaml` will add to the contents of `meta.yaml`. List values will be extended and string values will be concatenated. The proposed use case for this is to tweak/extend central recipes, such as those from conda-forge, with additional requirements while minimizing the actual changes to recipe files so as to avoid merge conflicts and source code divergence.

4.9.15 Partially clobbering recipes

As of conda-build 3.0, you can add a file named `recipe_clobber.yaml` in the same folder as your `meta.yaml` file. This file is considered to follow the same rules as `meta.yaml`, except that selectors and Jinja2 templates are not evaluated. Evaluation of selectors and Jinja2 templates will likely be added in future development.

Any contents in `recipe_clobber.yaml` will replace the contents of `meta.yaml`. This can be useful, for example, for replacing the source URL without copying the rest of the recipe into a fork.

4.9.16 Differentiating packages built with different variants

With only a few things supported, we could just add things to the filename, such as py27 for Python, or np111 for NumPy. Variants are meant to support the general case, and in the general case this is no longer an option. Instead, used variant keys and values are hashed using the SHA1 algorithm, and that hash is a unique identifier. The information that went into the hash is stored with the package in a file at `info/hash_input.json`. Packages only have a hash when there are any "used" variables beyond the ones that are already accounted for in the build string (py, np, etc). The takeaway message is that hashes will appear when binary compatibility matters, but not when it doesn't.

Currently, only the first 7 characters of the hash are stored. Output package names will keep the pyXY and npXYY, but may have added the 7-character hash. Your package names will look like:

`my-package-1.0-py27h3142afe_0.tar.bz2`

As of conda-build 3.1.0, this hashing scheme has been simplified. A hash will be added if all of these are true for any dependency:

- Package is an explicit dependency in build, host, or run deps.
- Package has a matching entry in `conda_build_config.yaml` which is a pin to a specific version, not a lower bound.
- That package is not ignored by ignore_version.

OR

- Package uses {{ compiler() }} Jinja2 function.

Since conflicts only need to be prevented within one version of a package, we think this will be adequate. If you run into hash collisions with this limited subspace, please file an issue on the conda-build issue tracker.

There is a CLI tool that just pretty-prints this JSON file for easy viewing:

```
conda inspect hash-inputs <package path>
```

This produces output such as:

```
{'python-3.6.4-h6538335_1': {'files': [],
                             'recipe': {'c_compiler': 'vs2015',
                                        'cxx_compiler': 'vs2015'}}}
```

4.9.17 Extra Jinja2 functions

Two especially common operations when dealing with these API and ABI incompatibilities are ways of specifying such compatibility, and of explicitly expressing the compiler to be used. Three new Jinja2 functions are available when evaluating `meta.yaml` templates:

- `pin_compatible('package_name', min_pin='x.x.x.x.x.x', max_pin='x', lower_bound=None, upper_bound=None)`: To be used as pin in run and/or test requirements. Takes package name argument. Looks up compatibility of named package installed in the build environment and writes compatible range pin for run and/or test requirements. Defaults to a semver-based assumption: `package_name >=(current version),<(next major version)`. Pass `min_pin` or `max_pin` a *Pinning expressions* . This will be enhanced as time goes on with information from ABI Laboratory.
- `pin_subpackage('package_name', min_pin='x.x.x.x.x.x', max_pin='x', exact=False)`: To be used as pin in run and/or test requirements. Takes package name argument. Used to refer to particular versions of subpackages built by parent recipe as dependencies elsewhere in that recipe. Can use either pinning expressions, or exact (including build string).
- `compiler('language')`: To be used in build requirements most commonly. Run or test as necessary. Takes language name argument. This is shorthand to facilitate cross-compiler usage. This Jinja2 function ties together 2 variant variables, `{language}_compiler` and `target_platform`, and outputs a single compiler package name. For example, this could be used to compile outputs targeting x86_64 and arm in one recipe, with a variant.

There are default "native" compilers that are used when no compiler is specified in any variant. These are defined in conda-build's jinja_context.py file. Most of the time, users will not need to provide compilers in their variants - just leave them empty and conda-build will use the defaults appropriate for your system.

4.9.18 Referencing subpackages

Conda-build 2.1 brought in the ability to build multiple output packages from a single recipe. This is useful in cases where you have a big build that outputs a lot of things at once, but those things really belong in their own packages. For example, building GCC outputs not only GCC, but also GFortran, g++, and runtime libraries for GCC, GFortran, and g++. Each of those should be their own package to make things as clean as possible. Unfortunately, if there are separate recipes to repack the different pieces from a larger, whole package it can be hard to keep them in sync. That's where variants come in. Variants, and more specifically the `pin_subpackage(name)` function, give you a way to refer to the subpackage with control over how tightly the subpackage version relationship should be in relation to other subpackages or the parent package. The following will output 5 conda packages.

`meta.yaml`:

```
package:
  name: subpackage_demo
  version: 1.0

requirements:
  run:
    - {{ pin_subpackage('subpackage_1') }}
    - {{ pin_subpackage('subpackage_2', max_pin='x.x') }}
    - {{ pin_subpackage('subpackage_3', min_pin='x.x', max_pin='x.x') }}
    - {{ pin_subpackage('subpackage_4', exact=True) }}


outputs:
  - name: subpackage_1
    version: 1.0.0
  - name: subpackage_2
    version: 2.0.0
  - name: subpackage_3
    version: 3.0.0
  - name: subpackage_4
    version: 4.0.0
```

Here, the parent package will have the following different runtime dependencies:

- subpackage_1 >=1.0.0,<2 (default uses `min_pin='x.x.x.x.x.x`, `max_pin='x'`, pins to major version with default >= current version lower bound)
- subpackage_2 >=2.0.0,<2.1 (more stringent upper bound)
- subpackage_3 >=3.0,<3.1 (less stringent lower bound, more stringent upper bound)
- subpackage_4 4.0.0 h81241af (exact pinning - version plus build string)

4.9.19 Compiler packages

On macOS and Linux, we can and do ship GCC packages. These will become even more powerful with variants since you can specify versions of your compiler much more explicitly and build against different versions, or with different flags set in the compiler package's activate.d scripts. On Windows, rather than providing the actual compilers in packages, we still use the compilers that are installed on the system. The analogous compiler packages on Windows run any compiler activation scripts and set compiler flags instead of actually installing anything.

Over time, conda-build will require that all packages explicitly list their compiler requirements this way. This is to both simplify conda-build and improve the tracking of metadata associated with compilers - localize it to compiler packages, even if those packages are doing nothing more than activating an already-installed compiler, such as Visual Studio.

Note also the `run_exports` key in `meta.yaml`. This is useful for compiler recipes to impose runtime constraints based on the versions of subpackages created by the compiler recipe. For more information, see the *Export runtime requirements* section of the `meta.yaml` docs. Compiler packages provided by Anaconda use the `run_exports` key extensively. For example, recipes that include the `gcc_linux-cos5-x86_64` package as a build time dependency (either directly, or through a `{{ compilers('c') }}` Jinja2 function) will automatically have a compatible libgcc runtime dependency added.

4.9.20 Compiler versions

Usually the newest compilers are the best compilers, but in some special cases you'll need to use older compilers.

For example, NVIDIA's CUDA libraries only support compilers that they have rigorously tested. Often the latest GCC compiler is not supported for use with CUDA. If your recipe needs to use CUDA, you'll need to use an older version of GCC.

There are special keys associated with the compilers. The key name of each special key is the compiler key name plus `_version`.

For example, if your compiler key is `c_compiler`, the version key associated with it is `c_compiler_version`. If you have a recipe for Tensorflow with GPU support, put a `conda_build_config.yaml` file alongside `meta.yaml`, with contents like:

```
c_compiler_version:     # [linux]
    - 5.4               # [linux]
cxx_compiler_version:   # [linux]
    - 5.4               # [linux]
```

Specify selectors so that this extra version information is not also applied to Windows and macOS. Those platforms have totally different compilers and could have their own versions if necessary.

It is not necessary to specify `c_compiler` or `cxx_compiler` because the default value (`gcc` on Linux) will be used. It is necessary to specify both `c` and `cxx` versions, even if they are the same, because they are treated independently.

By placing this file in the recipe, it will apply only to this recipe. All other recipes will default to the latest compiler.

Note: The version number you specify here must exist as a package in your currently configured channels.

4.9.21 Cross-compiling

The compiler Jinja2 function is written to support cross-compilers. This depends on setting at least 2 variant keys: `(language)_compiler` and `target_platform`. The target platform is appended to the value of `(language)_compiler` with the _ character. This leads to package names like `g++_linux-aarch64`. We recommend a convention for naming your compiler packages as: `<compiler name>_<target_platform>`.

Using a cross-compiler in a recipe would look like the following:

```
variants = {'cxx_compiler': ['g++'], 'target_platform': ['linux-cos5-x86_64', 'linux-
↪aarch64']}
```

and a `meta.yaml` file:

```
package:
    name: compiled-code
    version: 1.0

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

This assumes that you have created 2 compiler packages named `g++_linux-cos5-x86_64` and `g++_linux-aarch64` - all conda-build is providing you with is a way to loop over appropriately named cross-compiler toolchains.

4.9.22 Self-consistent package ecosystems

The compiler function is also how you could support a non-standard Visual Studio version, such as using VS 2015 to compile Python 2.7 and packages for Python 2.7. To accomplish this, you need to add the `{{ compiler('<language>') }}` to each recipe that will make up the system. Environment consistency is maintained through dependencies - thus it is useful to have the runtime be a versioned package with only one version being able to be installed at a time. For example, the `vc` package, originally created by Conda-Forge, is a versioned package (only one version can be installed at a time), and it installs the correct runtime package. When the compiler package imposes such a runtime dependency, then the resultant ecosystem is self-consistent.

Given these guidelines, consider a system of recipes using a variant like this:

```
variants = {'cxx_compiler': ['vs2015']}
```

The recipes include a compiler `meta.yaml` like this:

```
package:
    name: vs2015
    version: 14.0
build:
    run_exports:
        - vc 14
```

They also include some compiler-using `meta.yaml` contents like this:

```
package:
    name: compiled-code
    version: 1.0

requirements:
    build:
        # these are the same (and thus redundant) on windows, but different elsewhere
        - {{ compiler('c') }}
        - {{ compiler('cxx') }}
```

These recipes will create a system of packages that are all built with the VS 2015 compiler, and which have the vc package matched at version 14, rather than whatever default is associated with the Python version.

4.10 Conda-build CLI reference

Command-line interface (CLI) adds onto conda-build's functionality. CLI provides functions enabling you to convert packages between formats, render recipes, develop an index of packages, and more.

4.10.1 conda-build

4.10.2 conda convert

4.10.3 conda develop

4.10.4 conda index

4.10.5 conda inspect

4.10.6 conda inspect channels

4.10.7 conda inspect linkages

4.10.8 conda inspect objects

4.10.9 conda metapackage

4.10.10 conda render

4.10.11 conda skeleton

4.10.12 conda skeleton cpan

4.10.13 conda skeleton cran

4.10.14 conda skeleton luarocks

4.10.15 conda skeleton pypi

4.11 Adding Windows Start menu items

When a package is installed, it can add a shortcut to the Windows **Start** menu. Conda and conda-build handle this with the package menuinst, which currently supports only Windows. For instructions on using menuinst, see the menuinst wiki.

The easiest way to ensure that a package made with conda constructor does not install any menu shortcuts is to remove menuinst from the list of conda packages included. To do this, add the following to the `constuct.yaml` file:

```
exclude:
  - menuinst
```

4.12 Writing style guide

Follow these guidelines for submitting or editing conda-build documentation.

4.12.1 Audience

Identify who your audience is, their skill level, and how they can use the information.

4.12.2 Technical language

Match the level of technical language with the audience's level of proficiency. It's better to uderestimate the knowledge of your readers than overestimate it. Limit technical terms to those the user will encounter. If you must define a large number of terms, use a glossary to supplement definitions in the text.

4.12.3 Addressing the user

Use the active voice (e.g. Click this) and address users directly (write "you" rather than "the user"). When explaining an action, use the "command" form of the verb: "Choose an option from the menu and press Enter."

4.12.4 Format

See the *tutorial template* for the format. Provide descriptive task and subtask titles and do not number headings.

4.13 Tutorial template

- *Overview*
- *Who is this for?*
- *Before you start*
- *Tutorial tasks*
- *More information (optional)*

This document describes the steps for creating a tutorial for conda-build.

- *Copy the template*: https://github.com/conda/conda-build/tree/master/docs/source/resources/tutorial-template.rst.
- *Replace the italicized text with your tutorial content, following the* writing style guide.
- *Review other* tutorials *for additional guidance.*
- *Contact us at* documentation@anaconda.com *for help.*

4.13.1 Overview

Provide descriptions of the tutorial's:

- *Purpose*.
- *Benefits*.
- *Application*.

4.13.2 Who is this for?

- *Who is your audience*?
- *What skills or prior knowledge do they need*?
- *How will they use this tutorial*?

4.13.3 Before you start

Before you start, check the *Prerequisites*.

Provide descriptions of and links to requisite programs.

Glossary (optional)

If you're using several technical terms that your readers may be unfamiliar with, provide a glossary of key terms.

Glossary	Definition
Term 1	Term 1 defined
Term 2	Term 2 defined
Term 3	Term 3 defined

4.13.4 Tutorial tasks

- *Provide descriptive titles for each section*.
- *Identify the major tasks*.
- *Separate each major task into subtasks*.
- *Write a series of steps that walk the user through each subtask*.

4.13.5 More information (optional)

- *Provide links to related content*.
- *Add final notes for how to expand upon the tutorial*.

CHAPTER

FIVE

RELEASE NOTES

This information is drawn from the GitHub conda-build project changelog: https://github.com/conda/conda-build/blob/master/CHANGELOG.txt

5.1 3.18.11 (2019-11-01)

5.1.1 Enhancements

- Updated build.sh files of skeletons to be shellcheck clean, including test to lint future updates. Also added `set -o errexit -o pipefail` to build.sh files to make those settings transparent for better linting, while `errexit` was already the default used the call the build script.
- Corrected documentation on subpackage test requirements.
- Do not move work dir to work/work/
- Fixed a missing .lower() on two tar_xf related util functions
- Fixed `has_prefix detection` for Windows.
- `conda_build.inspect_pkg`: optimise use of fnmatch
- Do not consider .ignore files when searching with ripgrep
- Remove N*N os.lstat calls in `build_info_files_json_v1`

5.1.2 Contributors

- @msarahan
- @rrigdon
- @marcelotrevisani
- @rrigdon
- @soapy1
- @dbast
- @duncanmmacleod
- @beckermr
- @seanyen
- @AndrewAnnex

- @183amir
- @njzjz

5.2 3.18.10 (2019-10-14)

5.2.1 Enhancements

- Add the error message when an invalid pip dependency version expression is used.
- Conda skeleton pypi quoting just `version`, `summary`, and `description` of attributes with special characters.
- Set up CI Azure pipeline for Linux.
- Update cran skeleton to match supported optional licenses for license file derivation.
- Migrate Unittests to PyTest.
- Update script command on conda skeleton pypi to use `{{ PYTHON }} -m pip install . -vv`.
- Add a warning when a received a file on `RECIPE_PATH`.
- Refactor the skeletons/pypi.py get_package_metadata to be more modular.
- Add --suppress-variables switch to hide environment variables from console output.

5.2.2 Bug fixes

- Fix build of `.conda` packages enabled via `conda config --set conda_build.pkg_format 2`.
- Workaround for future deprecations of the SafeConfigParser and readfp of the same module.

5.2.3 Docs

- Remove bzip2 package from build toolkit description.

5.2.4 Contributors

- @msarahan
- @jakirkham
- @marcelotrevisani
- @duncanmmacleod
- @kinow
- @saraedum
- @jjhelmus
- @rrigdon
- @mingwandroid
- @asford

- @timsnyder
- @mcg1969
- @kaitietz
- @stuarteberg
- @isuruf
- @dbast
- @Bezier89

5.3 3.18.9 (2019-07-23)

5.3.1 Enhancements

- Add --use-channeldata argument to conda render/build.
- Extract the part in the skeletons pypi responsible to get the package metadata to a free function.
- Creat unittests for the get_package_metadata (skeletons/pypi.py) and for the new functions.

5.3.2 Bug fixes

- Limit threads to 61 on Windows.
- Do not use channeldata for run_exports unless --use-channeldata specified.
- Finalize top-level metadata if not present as an output.

5.3.3 Docs

- Add 3.18.7 release notes

5.3.4 Other

- Add disable_pip to FIELDS

5.3.5 Contributors

- @jjhelmus
- @rrigdon
- @Bezier89
- @jakirkham
- @marcelotrevisani

5.4 3.18.8 (2019-07-18)

5.4.1 Enhancements

- License_file can optionally be a yaml list

5.4.2 Bug fixes

- Fix readup of existing index.json in cache while extracting
- Fix spurious post build errors/warning message
- Merge channeldata from all urls

5.4.3 Contributors

- @msarahan
- @rrigdon
- @jjhelmus
- @isuruf
- @ddamiani

5.5 3.18.7 (2019-07-09)

5.5.1 Enhancements

- Update authorship for 3.18.7
- Add note on single threading for indexing during build
- Add in fallback for run_exports when channeldata not available
- Make pins for current_repodata additive - always newest, and pins are additions to that
- Limit indexing in build to using one thread
- Speed up by allowing empty run_exports entries in channeldata be valid results
- Bump conda-package-handling to 1.3+
- Add test for run_exports without channeldata
- Fallback to file-based run_exports if channeldata has no results
- Add Mozilla as valid license family
- Add in fallback for run_exports when channeldata not available
- Updated tutorials and resource documentation

5.5.2 Bug fixes

- Flake8 and test fixes from pytest deprecations
- Fix in render.py::_read_specs_from_package
- Fix for pkg_loc
- Fix conda debug output being suppressed

5.5.3 Contributors

- @msarahan
- @rrigdon
- @scopatz
- @mbargull
- @jakirkham
- @oleksandr-pavlyk

5.6 3.18.6 (2019-06-26)

5.6.1 Enhancements

- Package sha256 sums are included in index.html

5.6.2 Bug fixes

- Fix bug where package filenames were not included in the index.html

5.6.3 Contributors

- @rrigdon
- @jjhelmus

5.7 3.18.5 (2019-06-25)

5.7.1 Bug fixes

- Fix one more keyerror with missing timestamp data
- When indexing, allow .tar.bz2 files to use .conda cache, but not vice versa. This acts as a sanity check on the .conda files.
- Add build/rpaths_patcher to meta.yaml to allow switching between lief and patchelf for binary mangling

5.7.2 Contributors

- @mingwandroid
- @msarahan
- @csosborn

5.8 3.18.4 (2019-06-21)

5.8.1 Enhancements

- Channeldata reworked a bit to try to capture any available run_exports for all versions available

5.8.2 Bug fixes

- Make "timestamp" an optional field in conda index operations

5.8.3 Contributors

- @msarahan

5.9 3.18.3 (2019-06-20)

5.9.1 Enhancements

- Make VS2017 default Visual Studio
- Add hook for customizing the behavior of conda render
- Drop */usr* from CDT skeleton path
- Update cran skeleton to use m2w64 compilers for windows instead of toolchain. The linter is telling since long: Using toolchain directly in this manner is deprecated.

5.9.2 Bug fixes

- Update cran skeleton to not use toolchain for win
- Fix package_has_file so it supports .conda files (use cph)
- Fix package_has_file function for .conda format
- Fix off-by-one path trimming in prefix_files
- Disable overlinking checks when no files in the package have any shared library linkage
- Try to avoid finalizing top-level metadata twice
- Try to address permission errors on Appveyor and Azure by falling back to copy and warning (not erroring) if removing a file after copying fails
- Reduce the files inspected/loaded for channeldata, so that indexing goes faster

5.9.3 Deprecations

- The repodata2.json file is no longer created as part of indexing. It was not used by anything. It has been removed as an optimization. Its purpose was to explore namespaces and we'll bring its functionality back when we address that fully.

5.9.4 Contributors

- @mingwandroid
- @msarahan
- @rrigdon
- @soapy1
- @mariusvniekerk
- @jakirkham
- @dbast
- @duncanmmacleod

5.10 3.18.2 (2019-05-26)

5.10.1 Bug fixes

- Speed up post-link checks
- Fix activation not running during tests
- Improve indexing to show status better, and fix bug where size/hashes were being mixed up between .tar.bz2 and .conda files

5.10.2 Contributors

- @mingwandroid
- @msarahan
- @rrigdon

5.11 3.18.1 (2019-05-18)

5.11.1 Enhancements

- Rearrange steps in index.py to optimize away unnecessary work
- Restore parallel extract and hash in index operations

5.11.2 Contributors

- @msarahan

5.12 3.18.0 (2019-05-17)

5.12.1 Enhancements

- Set R_USER environment variable when building R packages
- Make Centos 7 default cdt distribution for linux-aarch64
- Bump default Python3 version to 3.7 for CI
- Build docs if any docs related file changes
- Add support for conda pkgv2 (.conda) format
- Add creation of "current_repodata.json" - like repodata.json, but only has the newest version of each file
- Change repodata layout to support .conda files. They live under the "packages.conda" key and have similar subkeys to their .tar.bz2 counterparts.
- Always show display actions, regardless of verbosity level
- Ignore registry autorun for all cmd.exe invocations
- Relax default pinning on r-base for benefit of noarch R packages
- Make conda index produce repodata_from_packages.json{,.bz2} which contains unpatched metadata
- Use a shorter environment prefix when testing on unix-like platforms
- Prevent pip from clobbering conda installed Python packages by populating .dist_info INSTALLER file

5.12.2 Bug fixes

- Allow build/missing_dso_whitelist section to be empty
- Make conda-debug honor custom channels passed using -c
- Do not attempt linkages inspection via lief if not installed
- Fix all lief related regressions brought in v3.17.x
- Fix ZeroDivisionError in ELF sections that have zero entries
- *binary_has_prefix_files* and *text_has_prefix_files* now override the automatically detected prefix replacement mode
- Handle special characters properly in pypi conda skeleton
- Build recipes in order of dependencies when passed to CB as directories
- Fix run_test script name for recipes with multiple outputs
- Fix recursion error with subpackages and build_id
- Avoid mutating global variable to fix tests on Windows
- Update CRAN license test case (replace r-ruchardet with r-udpipe)
- Update utils.filter_files to filter out generated .conda_trash files

- Replace stdlib glob with utils.glob. Latter supports recursion (**)

5.12.3 Docs

- Updated Sphinx theme to make notes and warnings more visible
- Added tutorial on building R-language packages using skeleton CRAN
- Add 37 to the list of valid values for CONDA_PY
- Corrected argparse rendering error
- Added tutorials section, reorganized content, and added a Windows tutorial
- Added Concepts section, removed extraneous content
- Added release notes section
- Reorganized sections
- Clarify to use 'where' on Windows and 'which' on Linux to inspect files in PATH
- Add RPATH information to compiler-tools documentation
- Improve the documentation on how to use the macOS SDK in build scripts.
- Document `conda build purge-all`.
- Fix user-guide index
- Add example for meta.yaml
- Updated theme
- Reorganized conda-build topics, updated link-scripts

5.12.4 Contributors

- @mingwandroid
- @msarahan
- @rrigdon
- @jjhelmus
- @nehaljwani
- @scopatz
- @Bezier89
- @rrigdon
- @isuruf
- @teake
- @jdblischak
- @bilderbuchi
- @soapy1
- @ESSS
- @tjd2002

- @tovrstra
- @chrisburr
- @katietz
- @hrzafer
- @zdog234
- @gabrielcnr
- @saraedum
- @uilianries
- @theultimate1
- @scw
- @spalmrot-tic

5.13 3.17.8 (2019-01-26)

5.13.1 Bug fixes

- Provide fallback from libarchive back to Python tarfile handling for handling tarfiles containing symlinks on windows

5.13.2 Other

- Rever support added for releasing conda-build

5.13.3 Contributors

- @msarahan
- @jjhelmus
- @scopatz
- @rrigdon
- @ax3l
- @rrigdon

5.14 3.17.7 (2019-01-16)

5.14.1 Bug fixes

- Respect context.offline setting #3328
- Don't write bytecode when building noarch: Python packages #3330
- Escape path separator in repl #3336

- Remove deprecated sudo statement from travis CI configuration #3338
- Fix running of test scripts in outputs #3343
- Allow overriding one key of zip_keys as long as length of group agrees #3344
- Fix compatibility with conda 4.6.0+ #3346
- Update centos 7 skeleton (CDT) URL #3350

5.14.2 Contributors

- @iainsgillis
- @isuruf
- @jjhelmus
- @nsoranzo
- @msarahan
- @qwhelan

INDEX

B

build string
    terminology, 12

C

canonical name
    terminology, 12

F

filename
    terminology, 12

N

name, *see* build string, *see* canonical name, *see* package name, *see* package version

P

package name
    terminology, 12
package spec, *see* package specification
package specification
    terminology, 13
package version
    terminology, 12

T

terminology
    build string, 12
    canonical name, 12
    filename, 12
    package name, 12
    package specification, 13
    package version, 12