Defining metadata (meta.yaml) — conda-build 3.21.4+10.ge458431a.dirty documentation

https://docs.conda.io/projects/conda-build/en/latest/resources/define-metadata.html

None Sun Jun, 20 23:53

- 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
  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.

Package section¶

Specifies package information.

Package name¶

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

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'
```

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¶

Source from svn¶

```
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.

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.

```
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.

```
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.

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.

```
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()`.

```
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`.

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.

```
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.

`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:

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.

Additionally, values can be set by including `=` followed by the desired value:

```
build:
  script_env:
   - MY_VAR=some value
```

Note

Inheriting environment variables can make it difficult for others to reproduce binaries from source with your recipe. Use this feature with caution or explicitly set values using the `=` syntax.

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
  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`.

If build and link dependencies need to impose constraints on the run environment but not necessarily pull in additional packages, then this can be done by altering the Run_constrained entries. In addtion to `weak` / `strong` `run_exports` which add to the `run` requirements, `weak_constrains` and `strong_constrains` add to the `run_constrained` requirements. With these, e.g., minimum versions of compatible but not required packages (like optional plugins for the linked dependency, or certain system attributes) can be expressed:

```
requirements:
  build:
    - build-tool                # has a strong run_constrained export
  host:
    - link-dependency           # has a weak run_constrained export
  run:
  run_constrained:
    # - system-dependency >=min  <-- implicitly added by build-tool
    # - optional-plugin >=min    <-- implicitly added by link-dependency
```

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++
```

You can also list the package the `run_exports` constraint is coming from using the `build/ignore_run_exports_from` section:

```
build:
  ignore_run_exports_from:
    - {{ compiler('cxx') }}
```

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:
```

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 }}
```

For example, let's say we have an environment that has package 'a' installed at version 1.0. If we install package 'b' that has a run_constrained entry of 'a>1.0', then conda would need to upgrade 'a' in the environment in order to install 'b'.

This is especially useful in the context of virtual packages, where the *run_constrained* dependency is not a package that conda manages, but rather a virtual package that represents a system property that conda can't change. For example, a package on linux may impose a *run_constrained* dependency on *__glibc>=2.12*. This is the version bound consistent with CentOS 6. Software built against glibc 2.12 will be compatible with CentOS 6. This *run_constrained* dependency helps conda tell the user that a given package can't be installed if their system glibc version is too old.

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 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.

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

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.

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
      - *.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.

```
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:

```
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`.

```
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`.

```
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.

```
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.

About section¶

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

```
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 or recipe directory. The value can be a single filename or a YAML list for multiple license files. Values can also point to directories with license information. Directory entries must end with a `/` suffix (this is to lessen unintentional inclusion of non-license files; all of the directory's contents will be unconditionally and recursively added).

```
about:
  license_file:
    - LICENSE
    - vendor-licenses/
```

Prelink Message File¶

Similar to the license file, the user can add prelink message files to the conda package.

```
about:
  prelink_message:
    - prelink_message_file.txt
    - folder-with-all-prelink-messages/
```

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.

```
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
```

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
```

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']
```

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.
arm64	True if the platform is macOS and the Python architecture is arm64.
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.
build_platform	The native subdir of the conda executable

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]
```

Note

To select multiple operating systems use the `or` statement. While it might be tempting to use `skip: True  # [win and osx]`, this will only work if the platform is both windows and osx simultaneously (i.e. never).

```
build:
   skip: True  # [win or osx]
```