Anaconda Project Documentation

Release 0.8.0rc5

Anaconda, Inc

Jul 09, 2021

Contents

Reproducible and executable project directories

Anaconda Project encapsulates data science projects and makes them easily portable. Project automates setup steps such as installing the right packages, downloading files, setting environment variables and running commands.

Project makes it easy to reproduce your work, share projects with others and run them on different platforms. It also simplifies deployment to servers. Anaconda projects run the same way on your machine, on another user's machine or when deployed to a server.

Traditional build scripts such as `setup.py` automate building the project—going from source code to something runnable—while Project automates running the project—taking build artifacts and doing any necessary setup before executing them.

You can use Project on Windows, macOS and Linux.

Project is supported and offered by Anaconda, Inc® and contributors under a 3-clause BSD license.

CHAPTER 1

Benefits of Project

- A `README` file that contains setup steps can become outdated, or users might not read it and then you have to help them diagnose problems. Project automates the setup steps so that the `README` file need only say "Type `anaconda-project run`."
- Project facilitates collaboration by ensuring that all users working on a project have the same dependencies in their Conda environments. Project automates environment creation and verifies that environments have the right versions of packages.
- You can run `os.getenv("DB_PASSWORD")` and configure Project to prompt the user for any missing credentials. This allows you to avoid including your personal passwords or secret keys in your code.
- Project improves reproducibility. Someone who wants to reproduce your analysis can ensure that they have the same setup that you have on your machine.
- Project simplifies deployment of your analysis as a web application. The configuration in `anaconda-project.yml` tells hosting providers how to run your project, so no special setup is needed when you move from your local machine to the web.

CHAPTER 2

How Project works

By adding an `anaconda-project.yml` configuration file to your project directory, a single `anaconda-project run` command can set up all dependencies and then launch the project. Running an Anaconda project executes a command specified in the `anaconda-project.yml` file, where you can also configure any arbitrary commands.

Project automates project setup by establishing all prerequisite conditions for the project's commands to execute successfully. These conditions could include:

- Creating a Conda environment that includes certain packages.
- Prompting the user for passwords or other configuration.
- Downloading data files.
- Starting extra processes such as a database server.

CHAPTER 3

Stability

Currently, the Project API and command-line syntax are subject to change in future releases. A project created with the current beta version of Project may always need to be run with that version of Project and not Project 1.0. When we think things are solid, we will switch from beta to version 1.0, and you will be able to rely on long-term interface stability.

3.1 Installation

Anaconda Project is included in Anaconda® versions 4.3.1 and later.

You can also get Project by installing Miniconda and then installing Project with this command:

```
conda install anaconda-project
```

Test your installation by running `anaconda-project` with the `version` option:

```
anaconda-project --version
```

A successful installation reports the version number.

3.2 Configuration

3.2.1 Environment variables

Anaconda Project has two modifiable configuration settings, both of which are currently controlled exclusively by environment variables.

`ANACONDA_PROJECT_ENVS_PATH` This variable provides a list of directories to search for environments to use in projects, and where to build them when needed. The format is identical to a standard `PATH` variable on the host operating system—a list of directories separated by `:` on Unix systems and `;` on Windows—except that empty entries are permitted. The paths are interpreted as follows:

- If a path is empty, it is interpreted as the default value `envs`.
- If a path is relative, it is interpreted relative to the root directory of the project itself (`PROJECT_DIR`). For example, a path entry `envs` is interpreted as
  - `$PROJECT_DIR/envs` (Unix)
  - `%PROJECT_DIR%\envs` (Windows)
- When searching for an environment, the directories are searched in left-to-right order.
- If an environment with the requested name is found nowhere in the path, one will be created as a subdirectory of the first entry in the path.

For example, given a Unix machine with

```
ANACONDA_PROJECT_ENVS_PATH=/opt/envs::envs2:/home/user/conda/envs
```

Then Anaconda Project will look for an environment named `default` in the following locations:

- `/opt/envs/default`
- `$PROJECT_DIR/envs/default`
- `$PROJECT_DIR/envs2/default`
- `/home/user/conda/envs/default`

If no such environment exists, one will be created as `/opt/envs/default`, instead of the default location of `$PROJECT_DIR/envs/default`.

`ANACONDA_PROJECT_READONLY_ENVS_POLICY` When an `anaconda-project.yml` specifies the use of an existing environment, but that environment is missing one or more of the requested packages, Anaconda Project attempts to remedy the deficiency by installing the missing packages. If the specified environment is *read-only*, however, such an installation would fail. The value of the environment variable `ANACONDA_PROJECT_READONLY_ENVS_POLICY` governs what action should be taken in such a case.

`fail` The attempt will fail, and a message returned indicating that the requested changes could not be made.

`clone` A clone of the read-only environment will be made, and additional packages will be installed into this cloned environment. Note that a clone will occur *only* if additional packages are required.

`replace` An entirely new environment will be created.

If this environment variable is empty or contains any other value than these, the `fail` behavior will be assumed. Note that for `clone` or `replace` to succeed, a writable environment location must exist somewhere in the `ANACONDA_PROJECT_ENVS_PATH` path.

3.2.2 Read-only environments

On some systems, it is desirable to provide Anaconda Project access to one or more *read-only* environments. These environments can be centrally managed by administrators, and will speed up environment preparation for users that elect to use them.

On Unix, a read-only environment is quite easy to enforce with standard POSIX permissions settings. Unfortunately, our experience on Windows systems suggests it is more challenging to enforce. For this reason, we have adopted a simple approach that allows environments to be explicitly marked as read-only with a flag file:

- If a file called `.readonly` is found in the root of an environment, that environment will be considered read-only.
- If a file called `.readonly` is found in the *parent* of an environment directory, the environment will be considered read-only.

- An attempt is made to write a file `var/cache/anaconda-project/status` within the environment, creating the subdirectories as needed. If successful, the environment is considered read-write; otherwise, it is considered read-only.

This second test is particularly useful when centrally managing and entire directory of environments. With a single `.readonly` flag file, all new environments created within that directory will be treated as read-only. Of course, for the best protection, POSIX or Windows read-only permissions should be applied nevertheless.

3.3 User guide

3.3.1 Concepts

Project

A project is a folder that contains an `anaconda-project.yml` configuration file together with scripts, notebooks and other files.

You can make any folder into a project by adding a configuration file named `anaconda-project.yml` to the folder. The configuration file can include the following sections:

- commands
- variables
- services
- downloads
- packages or dependencies
- channels
- env_specs

Data scientists use projects to encapsulate data science projects and make them easily portable. A project is usually compressed into a .tar.bz2 file for sharing and storing.

Anaconda Project automates setup steps, so that data scientists that you share projects with can run your project with a single command—`anaconda-project run`.

Configuration files

Projects are affected by 3 configuration files:

- `anaconda-project.yml`—Contains information about a project to be shared across users and machines. If you use source control, put `anaconda-project.yml` into your system.

- `anaconda-project-local.yml`—Contains your local configuration state, which you do not want to share with others. Put this file into .gitignore, .svnignore or the equivalent in your source control system.
- `anaconda-project-lock.yml`—Optional. Contains information needed to lock your package dependencies at specific versions. Put this file into source control along with `anaconda-project.yml`. For more information on `anaconda-project-lock.yml`, see *Reference*.

To modify these files, use Project commands, Anaconda Navigator, or any text editor.

Environment variables

Anything in the "variables" section of an `anaconda-project.yml` file is considered to be an environment variable needed by your project.

EXAMPLE: The variables section of an `anaconda-project.yml` file that specifies 2 variables:

```
variables:
  - AMAZON_EC2_USERNAME
  - AMAZON_EC2_PASSWORD
```

When a user runs your project, Project asks them for values to assign to these variables.

In your script, you can use `os.getenv()` to obtain these variables. This is a much better option than hardcoding passwords into your script, which can be a security risk.

Comparing Project to conda env and environment.yml

Project has similar functionality to the `conda env` command and the `environment.yml` file, but it may be more convenient. The advantage of Project for environment handling is that it performs conda operations and records them in a configuration file for reproducibility, all in one step.

EXAMPLE: The following command uses conda to install Bokeh and adds `bokeh=0.11` to an environment spec in `anaconda-project.yml`:

```
anaconda-project add-packages bokeh=0.11
```

The effect is comparable to adding the environment spec to `environment.yml`. In this way, the state of your current conda environment and your configuration to be shared with others will not get out of sync.

Project also automatically sets up environments for other users when they type `anaconda-project run` on their machines. They do not have to separately create, update or activate environments before they run the code. This may be especially useful when you change the required dependencies. With `conda env`, users may forget to rerun it and update their packages, while `anaconda-project run` automatically adds missing packages every time.

In addition to creating environments, Project can perform other kinds of setup, such as adding data files and running a database server. In that sense, it is a superset of `conda env`.

3.3.2 Getting started

This getting started guide walks you through using Anaconda Project for the first time.

After completing this guide, you will be able to:

- Create a project containing a Bokeh app.
- Run the project with a single command.
- Package and share the project.

If you have not yet installed and started Project, follow the *Installation instructions*.

For more information on Bokeh, see Welcome to Bokeh.

Creating a project containing a Bokeh app

To create a project called "clustering_app":

1. Open a Command Prompt or terminal window.
2. Create a directory called `clustering_app`, switch to it and then initialize the project:

```
$ mkdir clustering_app
$ cd clustering_app
$ anaconda-project init
Project configuration is in /User/Anaconda/My Anaconda Projects/clustering_app/
↪anaconda-project.yml
```

3. Inside the `clustering_app` project directory, create and save a file named `main.py` that contains the code from the Bokeh clustering example.
4. Add the packages that the Bokeh clustering demo depends on:

```
anaconda-project add-packages python=3.5 bokeh=0.12.4 numpy=1.12.0 scikit-learn=0.
↪18.1
```

5. Tell Project about the Bokeh app:

```
anaconda-project add-command plot .
```

   NOTE: By default, Bokeh looks for the file `main.py`, so you do not need to include this in the command string after the "plot" command name.

6. When prompted, type `B` for Bokeh app:

```
Is `plot` a (B)okeh app, (N)otebook, or (C)ommand line? B
Added a command 'plot' to the project.
Run it with `anaconda-project run plot`.
```

7. Run your new project:

```
anaconda-project run
```

   NOTE: If your project included more than one command, you would need to specify which command to run. For more information, see *Running a project*.

   A browser window opens, displaying the clustering app.

Sharing your project

To share this project with a colleague:

1. Archive the project:

```
anaconda-project archive clustering.zip
```

2. Send the archive file to your colleague.

You can also share a project by uploading it to Anaconda Cloud. For more information, see *Sharing a project*.

Running your project

Anyone with Project—your colleague or someone who downloads your project from Cloud—can run your project by unzipping the project archive file and then running a single command, without having to do any setup:

```
anaconda-project run
```

NOTE: If your project contained more than one command, the person using your project would need to specify which command to run. For more information, see *Running a project*.

Project downloads the data, installs the necessary packages and runs the command.

Next steps

- Learn more about *what you can do in Project*, including how to *download data* with your project and how to *configure your project with environment variables*.
- Learn more about *the anaconda-project.yml format*.

3.3.3 Tasks

Creating a project

1. Create a project directory:

   ```
   anaconda-project init --directory directory-name
   ```

   NOTE: Replace `directory-name` with the name of your project directory.

   EXAMPLE: To create a project directory called "iris":

   ```
   $ cd /home/alice/mystuff
   $ anaconda-project init --directory iris
   Create directory '/home/alice/mystuff/iris'? y
   Project configuration is in /home/alice/mystuff/iris/anaconda-project.yml
   ```

   You can also turn any existing directory into a project by switching to the directory and then running `anaconda-project init` without options or arguments.

2. OPTIONAL: In a text editor, open `anaconda-project.yml` to see what the file looks like for an empty project. As you work with your project, the `anaconda-project` commands you use will modify this file.

As of version 0.10.0 `anaconda-project init` will initialize an empty environment. No packages will be listed in the `pakcages:` key. To replicate this behavior on older versions run:

```
anaconda-project init --empty-environment
```

Working with packages

The `anaconda-project.yml` file enables specification of required packages and multiple Conda environments, referred to as `env_specs`.

For example the following `anaconda-project.yml` file will install `python` version 3.8, and latest version `pandas` and `notebook` into the default environment when you execute `anaconda-project prepare` on the command line.

```
name: ExampleProject

packages:
  - python=3.8
  - notebook
  - pandas

env_specs:
  default: {}
```

When `anaconda-project prepare` is run a new environment is created called `default` in the `envs` subdirectory of your project. See *Configuration* to change the default location of the Conda environments.

Adding packages

To add packages to your project that are not yet in your `packages:` list there are two approaches.

1. From within your project directory, run:

   ```
   anaconda-project add-packages package1 package2
   ```

   NOTE: Replace `package1` and `package2` with the names of the packages that you want to include. You can specify as many packages as you want.

   EXAMPLE: To add the packages hvplot=0.7 and dask:

   ```
   $ anaconda-project add-packages hvplot=0.7 dask
   Collecting package metadata (current_repodata.json): ...working... done
   Solving environment: ...working... done
   ...
   Executing transaction: ...working... done
   Using Conda environment /Users/adefusco/Desktop/iris/envs/default.
   Added packages to project file: hvplot=0.7, dask.
   ```

2. Instead of using the `add-packages` command you can edit the `anaconda-project.yml` file directly using any text editor and add package names to the `packages:` list. To complete the installation of these new packages into your activate environment run `anaconda-project prepare` on the command line after saving the file.

In addition to adding Conda packages as shown above Pip packages can be specified using the `--pip` flag:

```
anaconda-project add-packages --pip package1 package2
```

NOTE: Replace `package1` and `package2` with the names of the packages that you want to include. You can specify as many packages as you want.

EXAMPLE: To add the `requests` package to the default environment:

```
$ anaconda-project add-packages --pip requests
Collecting requests
  Using cached requests-2.25.1-py2.py3-none-any.whl (61 kB)
Requirement already satisfied: certifi>=2017.4.17 in ./envs/default/lib/python3.8/
↪site-packages (from requests) (2020.12.5)
Collecting idna<3,>=2.5
  Using cached idna-2.10-py2.py3-none-any.whl (58 kB)
Collecting chardet<5,>=3.0.2
  Using cached chardet-4.0.0-py2.py3-none-any.whl (178 kB)
```

(continues on next page)

(continued from previous page)

```
Collecting urllib3<1.27,>=1.21.1
  Using cached urllib3-1.26.4-py2.py3-none-any.whl (153 kB)
Installing collected packages: urllib3, idna, chardet, requests
Successfully installed chardet-4.0.0 idna-2.10 requests-2.25.1 urllib3-1.26.4
Using Conda environment /Users/adefusco/Desktop/testproj/envs/default.
Added packages to project file: requests.
```

Optionally, you can edit the `anaconda-project.yml` file to add packages using the `pip:` key within the `packages:` list. For example,

```
name: ExampleProject

packages:
  - python=3.8
  - notebook
  - pandas
  - pip:
    - requests

env_specs:
  default: {}
```

Then run `anaconda-project prepare` to install the new packages into the environment.

Removing packages

To remove packages from the `packages:` list run:

```
anaconda-project remove-packages package1 package2
```

NOTE: Replace `package1` and `package2` with the names of the packages that you want to include. You can specify as many packages as you want.

EXAMPLE: To remove the package hvplot:

```
$ anaconda-project remove-packages hvplot
Using Conda environment /Users/adefusco/Desktop/testproj/envs/default.
Removed packages from project file: hvplot.
```

EXAMPLE: To remove the `requests` pip package from the default environment:

```
$ anaconda-project remove-packages --pip requests
Found existing installation: requests 2.25.1
Uninstalling requests-2.25.1:
  Successfully uninstalled requests-2.25.1
Using Conda environment /Users/adefusco/Desktop/testproj/envs/default.
Removed packages from project file: requests.
```

Pip package specifications

Pip packages can specified in a number of ways.

- From PyPI (or other indexes)
- Direct URL to the package archive

- Revision Control services (for example git and svn)

To install a package from a revision control service:

```
anaconda-project add-packages --pip git+<protocol>://<revision-control-domain>/
↪<repository.git>[version-branch]#egg=<package-name>
```

Where

- `<protocol>` is the web protocol of the domain: i.e, `http` or `https`
- `<revision-control-domain>` is the URL of the service: i.e. `github.com`
- `<repository.git>` is the name of the revision control repository, you can include the branch name or release tag here.
- `[version-branch]` optionally install a specific version or branch of the repository
- `<package>` is the name of the package as declared in `setup.py`

NOTE: It is required that you use `#egg=<package>` to install a revision control hosted package. This is considered best practice by pip and allows the pip dependency solver to correctly identify the package if it is a dependency of another package in your project.

EXAMPLE: Add the tranquilizer package to your project directly from Github:

```
$ anaconda-project add-packages --pip git+https://github.com/continuumio/tranquilizer.
↪git@0.5.0#egg=tranquilizer
Collecting tranquilizer
Cloning https://github.com/continuumio/tranquilizer.git (to revision 0.5.0) to /
↪private/var/folders/lk/s__7f9fx15x_zrw6q5xkmm500000gp/T/pip-install-5ncd7pbt/
↪tranquilizer_d037aa7b85d048c1acd4e2f0044c4cea
Using Conda environment /Users/adefusco/Desktop/testproj/envs/default.
Added packages to project file: git+https://github.com/continuumio/tranquilizer.git@0.
↪5.0#egg=tranquilizer.
```

Alternatively for github you can use the URL of the repository archive. For example, to install from the master branch of tranquilizer:

```
$ anaconda-project add-packages --pip https://github.com/continuumio/tranquilizer/
↪archive/master.zip#egg=tranquilizer
Collecting tranquilizer
Downloading https://github.com/continuumio/tranquilizer/archive/master.zip
Using Conda environment /Users/adefusco/Desktop/testproj/envs/default.
Added packages to project file: https://github.com/continuumio/tranquilizer/archive/
↪master.zip#egg=tranquilizer.
```

Downloading data into a project

Often data sets are too large to keep locally, so you may want to download them on demand.

To set up your project to download data:

1. From within your project directory, run:

   ```
   anaconda-project add-download env_var URL
   ```

   NOTE: Replace `env_var` with a name for an environment variable that Anaconda Project will create to store the path to your downloaded data file. Replace `URL` with the URL for the data to be downloaded.

   Anaconda Project downloads the data file to your project directory.

EXAMPLE: The following command downloads the `iris.csv` data file from a GitHub repository into the "iris" project, and stores its new path in the environment variable IRIS_CSV:

```
$ anaconda-project add-download IRIS_CSV https://raw.githubusercontent.com/bokeh/
↪bokeh/f9aa6a8caae8c7c12efd32be95ec7b0216f62203/bokeh/sampledata/iris.csv
File downloaded to /home/alice/mystuff/iris/iris.csv
Added https://raw.githubusercontent.com/bokeh/bokeh/
↪f9aa6a8caae8c7c12efd32be95ec7b0216f62203/bokeh/sampledata/iris.csv to the␣
↪project file.
```

2. OPTIONAL: In a text editor, open `anaconda-project.yml` to see the new entry in the downloads section.

Working with commands

Run all of the commands on this page from within the project directory.

Adding a command to a project

A project contains some sort of code, such as Python files, which have a `.py` extension.

You could run your Python code with the command:

```
python file.py
```

NOTE: Replace `file` with the name of your file.

However, to gain the benefits of Anaconda Project, use Project to add code files to your project:

1. Put the code file, application file, or notebook file into your project directory.
2. Add a command to run your file:

   ```
   anaconda-project add-command name "command"
   ```

   NOTE: Replace `name` with a name of your choosing for the command. Replace `command` with the command string.

   EXAMPLE:: To add a command called "notebook" that runs the IPython notebook `mynotebook.ipynb`:

   ```
   anaconda-project add-command notebook mynotebook.ipynb
   ```

   EXAMPLE: To add a command called "plot" that runs a Bokeh app located outside of your project directory:

   ```
   anaconda-project add-command plot app-path-filename
   ```

NOTE: Replace `app-path-filename` with the path and filename of the Bokeh app. By default, Bokeh looks for the file `main.py`, so if your app is called `main.py`, you do not need to include the filename.

3. When prompted for the type of command, type:
   - `B` if the command string is a Bokeh app to run.
   - `N` if the command string is a Notebook to run.
   - `C` if the command string is a Command line instruction to run, such as using Python to run a Python .py file.

   EXAMPLE: To add a command called "hello" that runs `python hello.py`:

```
$ anaconda-project add-command hello "python hello.py"
Is `hello` a (B)okeh app, (N)otebook, or (C)ommand line? C
Added a command 'hello' to the project. Run it with
`anaconda-project run hello`.
```

4. OPTIONAL: In a text editor, open `anaconda-project.yml` to see the new command listed in the commands section.

Specifying multi-line commands

Commands added to the `anaconda-project.yml` file can span multiple lines of execution by using the YAML `|` string specifier. For example a single command can be defined in the `anaconda-project.yml` file to run multiple linting tools.

```
commands:
  unix: |
    yapf --in-place
    flake8
    pep256
```

Using commands that need different environments

You can have multiple conda environment specifications in a project, which is useful if some of your commands use a different version of Python or otherwise have distinct dependencies. Add these environment specs with `anaconda-project add-env-spec`.

Using commands to automatically start processes

Project can automatically start processes that your commands depend on. Currently it only supports starting Redis, for demonstration purposes.

To see Project automatically start the Redis process:

```
anaconda-project add-service redis
```

More types of services will be supported soon. If there are particular services that you would find useful, *let us know*.

Viewing a list of commands in a project

To list all of the commands in a project:

```
anaconda-project list-commands
```

EXAMPLE:

```
$ anaconda-project list-commands
Commands for project: /home/alice/mystuff/iris

Name       Description
====       ===========
hello      python hello.py
plot       Bokeh app iris_plot
showdata   python showdata.py
```

Running a project command

Running a project command is the same as *Running a project*.

Working with environment variables

Run all of the commands on this page from within the project directory.

Anaconda Project sets some environment variables automatically:

- PROJECT_DIR specifies the location of your project directory.
- CONDA_ENV_PATH is set to the file system location of the current conda environment.
- PATH includes the binary directory from the current conda environment.

These variables always exist and can always be used in your Python code.

Using variables in scripts

Use Python's os.getenv() function to obtain variables from within your scripts.

EXAMPLE: The following script, called `showdata.py`, prints out data:

```
import os
import pandas as pd

project_dir = os.getenv("PROJECT_DIR")
```

(continues on next page)

(continued from previous page)

```
env = os.getenv("CONDA_DEFAULT_ENV")
iris_csv = os.getenv("IRIS_CSV")

flowers = pd.read_csv(iris_csv)

print(flowers)
print("My project directory is {} and my conda environment is {}".format(project_dir,␣
↪env))
```

If you tried to run this example script with `python showdata.py`, it would not work if pandas was not installed and the environment variables were not set.

Adding a variable

If a command needs a user-supplied parameter, you can require—or just allow—users to provide values for these before the command runs.

NOTE: Encrypted variables such as passwords are treated differently from other custom variables. See *Adding an encrypted variable*.

1. Add the unencrypted variable to your project:

   ```
   anaconda-project add-variable VARIABLE
   ```

   NOTE: Replace VARIABLE with the name of the variable that you want to add.

   EXAMPLE: To add a variable called COLUMN_TO_SHOW:

   ```
   anaconda-project add-variable COLUMN_TO_SHOW
   ```

2. OPTIONAL: In a text editor, open `anaconda-project.yml` to see the new variable listed in the variables section.
3. OPTIONAL: Use the command `anaconda-project list-variables` to see the new variables listed.
4. Include the new variable in your script in the same way as you would for any other variable.

The first time a user runs your project, they are prompted to provide a value for your custom variable. On subsequent runs, the user will not be prompted.

All environment variables are available for jinja2 templating as shown in the *HTTP Commands* section.

Adding an encrypted variable

Use variables for passwords and other secret information so that each user can input their own private information.

Encrypted variable values are kept in the system keychain, while other variable values are kept in the `anaconda-project-local.yml` file. In all other respects, working with encrypted variables is the same as for unencrypted variables.

Any variable ending in `_PASSWORD`, `_SECRET`, or `_SECRET_KEY` is automatically encrypted.

To create an encrypted variable:

```
anaconda-project add-variable VARIABLE_encrypt-flag
```

NOTE: Replace VARIABLE with the name of the variable that you want to add. Replace `_encrypt-flag` with `_PASSWORD`, `_SECRET` or `_SECRET_KEY`.

EXAMPLE: To create an encrypted variable called `DB_PASSWORD`:

```
anaconda-project add-variable DB_PASSWORD
```

Adding a variable with a default value

You can set a default value for a variable, which is stored with the variable in `anaconda-project.yml`. If you set a default, users are not prompted to provide a value, but they can override the default value if they want to.

To add a variable with a default value:

```
anaconda-project add-variable --default=default_value VARIABLE
```

NOTE: Replace `default_value` with the default value to be set and `VARIABLE` with the name of the variable to create.

EXAMPLE: To add the variable `COLUMN_TO_SHOW` with the default value `petal_width`:

```
anaconda-project add-variable --default=petal_width COLUMN_TO_SHOW
```

If you or a user sets the variable in `anaconda-project-local.yml`, the default is ignored. However, you can unset the local override so that the default is used:

```
anaconda-project unset-variable VARIABLE
```

NOTE: Replace VARIABLE with the variable name.

EXAMPLE: To unset the `COLUMN_TO_SHOW` variable:

```
anaconda-project unset-variable COLUMN_TO_SHOW
```

Changing a variable's value

The variable values entered by a user are stored in the user's `anaconda-project-local.yml` file. To change a variable's value in the user's file:

```
anaconda-project set-variable VARIABLE=value
```

NOTE: Replace `VARIABLE` with the variable name and `value` with the new value for that variable.

EXAMPLE: To set `COLUMN_TO_SHOW` to `petal_length`:

```
anaconda-project set-variable COLUMN_TO_SHOW=petal_length
```

Removing a variable's value

Use the `unset-variable` command to remove the value that has been set for a variable. Only the value is removed. The project still requires a value for the variable in order to run.

Removing a variable

Use the `remove-variable` command to remove the variable from `anaconda-project.yml` so that the project no longer requires the variable value in order to run.

Running a project

Run all of the commands on this page from within the project directory.

To run a project:

1. If necessary, extract the files from the project archive file—.zip, .tar.gz or .tar.bz2.
2. If you do not know the exact name of the command you want to run, *list the commands* in the project.
3. If there is only one command in the project, run:

```
anaconda-project run
```

4. If there are multiple commands in the project, include the command name:

```
anaconda-project run command-name
```

   NOTE: Replace `command-name` with the actual command name.

   EXAMPLE: To run a command called "showdata", which could download data, install needed packages and run the command:

```
anaconda-project run showdata
```

5. For a command that runs a Bokeh app, you can include options for `bokeh serve` in the run command.

   EXAMPLE: The following command passes the `--show` option to the `bokeh serve` command, to tell Bokeh to open a browser window:

```
anaconda-project run plot --show
```

When you run a project for the first time, there is a short delay as the new dedicated project is created, and then the command is executed. The command will run much faster on subsequent runs because the dedicated project is already created.

In your project directory, you now have an `envs` subdirectory. By default every project has its own packages in its own sandbox to ensure that projects do not interfere with one another.

Cleaning a project

Your projects contain files that Anaconda Project creates automatically, such as any downloaded data and the `envs/default` directory.

Use the `clean` command to remove such files and make a clean, reproducible project.

Run the following command from within the project directory:

```
anaconda-project clean
```

Project removes automatically created files and downloaded data.

To restore these files:

- *Prepare the project*.

  OR

- *Run the project*.

Preparing a project

When you run a project, Anaconda Project automatically generates certain files and downloads necessary data. The `prepare` command allows you to initiate that process without running the project.

To prepare a project, run the `prepare` command from within your project directory:

```
anaconda-project prepare
```

Creating a project archive

To share a project with others, you likely want to put it into an archive file, such as a .zip file. Anaconda Project can create .zip, .tar.gz and .tar.bz2 archives. The archive format matches the file extension that you provide.

Excluding files from the archive

The `anaconda-project archive` command automatically omits the files that Project can reproduce automatically, which includes the `envs/` directory and any downloaded data files defined in the *downloads* section of the `anaconda-project.yml` file.

See *Packagaging Environments* below to bundle Conda environments in the archive.

The archive also excludes `anaconda-project-local.yml`, which is intended to hold local configuration state only.

To manually exclude any other files that you do not want to be in the archive, create a `.projectignore` file or a `.gitignore` file.

Note: If you anticipate that this project will be managed as a Git repository, use of `.gitignore` is preferred over `.projectignore`. However, use of `.gitignore` outside of a Git repository is not supported.

Creating the archive file

To create a project archive, run the following command from within your project directory:

```
anaconda-project archive filename.zip
```

NOTE: Replace `filename` with the name for your archive file. If you want to create a .tar.gz or .tar.bz2 archive instead of a zip archive, replace `zip` with the appropriate file extension.

EXAMPLE: To create a zip archive called “iris”:

```
anaconda-project archive iris.zip
```

Project creates the archive file.

If you list the files in the archive, you will see that automatically generated files are not listed.

EXAMPLE:

```
$ unzip -l iris.zip
Archive:  iris.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
       16  06-10-2016 10:04   iris/hello.py
      281  06-10-2016 10:22   iris/showdata.py
      222  06-10-2016 09:46   iris/.projectignore
     4927  06-10-2016 10:31   iris/anaconda-project.yml
      557  06-10-2016 10:33   iris/iris_plot/main.py
---------                     -------
     6003                     5 files
```

Extracting the archive file

Anaconda Project archives can be extracted using packages provided by the OS or using the `anaconda-project unarchive` command.

The `unarchive` command can extract bundles in any of the supported formats (`.zip`, `.tar.gz`, and `.tar.bz2`):

```
anaconda-project unarchive <bundle>
```

Experimental: Packaging environments

Available since version 0.10.0

There are cases where it may be preferable to package the Conda environments directly into the archive. For example, you may want to support uses where the target system cannot connect to the repository to download and install packages.

To bundle the environments into the archive use the `--pack-envs` flag. This will utilize conda-pack to add each `env_spec` to the Anaconda Project bundle.

With the `--pack-envs` the `prepare` command is run automatically to ensure that all `env_specs` are up-to-date before building the bundle.

Note: When using `--pack-envs` your Anaconda Project bundles may be very large.

The bundle can be extracted using either `anaconda-project unarchive` or OS packages for Zip, `tar.gz`, and `tar.bz2` files.

If a pack-envs bundle is extracted on a platform (Mac, Linux, Windows) that does not match the platform used to create the bundle the `env_specs` will be re-created when you run `anaconda-project prepare` or `anaconda-project run`.

Sharing a project

To share a project with other people:

1. *Archive* the project.
2. Send the file to another user—for example, by email, by copying the file to a shared network location, and so on.

OR

Upload the file to Anaconda Cloud by running the following command from within the project directory:

```
anaconda-project upload
```

NOTE: You need a free Cloud account to upload projects to Cloud.

3. The user retrieves the archive file and *runs the project*.

Creating Docker Images

Available since version 0.10.0

Use the `dockerize` command to create a Docker image from the project. Images created from Anaconda Projects are configured to execute a single *defined command* in the `anaconda-project.yml` file chosen at build time.

The `dockerize` command uses source-to-image (s2i) to build Docker images using the s2i-anaconda-project builder images that have been uploaded to Docker Hub.

Images built with `dockerize` will have a fully prepared `env_spec` for the desired command and expose port 8086 if the command listens for HTTP requests.

Prerequisites

In order to utilize the `dockerize` command you will need to have Docker and source-to-image (s2i) installed.

- Docker
- source-to-image

You can install s2i using Conda

```
conda install -c ctools source-to-image
```

Quickstart

1. Build a docker image to run a supplied command

```
anaconda-project dockerize --command <command-name> -t <image name>
```

2. Run the Docker image and publish port 8086

```
docker run -p 8086:8086 <image name>
```

It is necessary to add `-p 8086:8086` in order to publish port 8086 from the anaconda-project container out to the host. The second entry in the `-p` flag must be `8086` while the first entry can be any valid unused port on the host. See the Docker container networking docs for more details.

Build Docker images

By default running the `dockerize` command will create a Docker image to execute the *default* command.

The *default* command is determined in the following order

1. The command named `default`

2. The first command listed in the project file if no command is named `default`

The `s2i-anaconda-project` builder images have Miniconda and `anaconda-project` installed. When the `dockerize` command is run the following steps are performed.

1. The project is archived to a temporary directory to ensure that files listed in your `.projectignore` (including the local `envs` directory) are not copied into the Docker image.
2. The `s2i build` command is run from the temporary directory to construct a new Docker image from the builder image.

The steps in the `s2i build` process are

1. The temporary project directory is added to the image.
2. The s2i assemble script is run to prepare the `env_spec` for the desired command.
3. Conda clean is run to reduce the size of the output Docker image.

Options

The `dockerize` command accepts several optional arguments

`--command` The named command to execute in the `RUN` layer of the Docker image. Default: `default` See the HTTP commands section below.

`-t` or `--tag` The name of the output Docker image in the format `name:tag`. By default Default: "<project-name>:latest", where <project-name> is taken from the name tag in the anaconda-project.yml file.

`--builder-image` The s2i builder image name to use. Default: `conda/s2i-anaconda-project-ubi8` By default this is image is pulled from DockerHub when `dockerize` is run. See the Custom Builder Image section below to construct your own builder images.

s2i build arguments Any further arguments or those supplied after `--` will be given to the `s2i build` command. See the s2i build documentation for available build flags.

Builder images

The default builder image for `anaconda-project dockerize` is `conda/s2i-anaconda-project-ubi8`. To see other available builder images on DockerHub click here.

HTTP options

When the docker image is run the s2i run script is executed with the supplied command. The full run command is

```
anaconda-project run $CMD --anaconda-project-port 8086 --anaconda-project-address 0.0.
↪0.0 --anaconda-project-no-browser --anaconda-project-use-xheaders
```

This ensures that the command communicates over port 8086 if it supports the *HTTP Commands*.

If your desired command is not an HTTP command or you wish not to use the Jinja2 template features you must add `supports_http_options: false` to the command specification in the `anaconda-project.yml` file. When `supports_http_options` is set to `false` the HTTP arguments are completely ignored when the command is executed.

Configuring Conda

In addition to the channel configuration available in the `anaconda-project.yml` file you may need to supply custom Conda configuration parameters in order to build the Docker image.

To provide a custom Conda configuration, place a `.condarc` file at the top-level of your project directory.

For example, you can use the `.condarc` to configure access to Anaconda Team Edition or Anaconda Commercial Edition.

Custom builder images

If you want to customize the builder images you can clone the s2i-anaconda-project repository, build the images. The custom builder images can be provided to `anaconda-project dockerize` using the `--builder-image` flag.

3.3.4 Reference

The `anaconda-project` command works with *project directories*, which can contain scripts, notebooks, data files, and anything that is related to your project.

Any directory can be made into a project by adding a configuration file named `anaconda-project.yml`.

`.yml` files are in the YAML format and follow the YAML syntax.

TIP: Read more about YAML syntax at http://yaml.org/start.html

TIP: You may want to go through the anaconda-project tutorial before digging into the details in this document.

`anaconda-project.yml`, `anaconda-project-local.yml`, `anaconda-project-lock.yml`

Anaconda projects are affected by three configuration files, `anaconda-project.yml`, `anaconda-project-local.yml`, and `anaconda-project-lock.yml`.

The file `anaconda-project.yml` contains information about a project that is intended to be shared across users and machines. If you use source control, the file `anaconda-project.yml` should probably be put in source control.

The file `anaconda-project-local.yml`, on the other hand, goes in `.gitignore` (or `.svnignore` or equivalent), because it contains your local configuration state that you do not want to share with others.

The file `anaconda-project-lock.yml` is optional and contains information needed to lock your package dependencies at specific versions. This "lock file" should go in source control along with `anaconda-project.yml`.

These files can be manipulated with `anaconda-project` commands, or with Anaconda Navigator, or you can edit them with any text editor.

Commands and Requirements

In the `anaconda-project.yml` file you can define *commands* and *requirements* that the commands need in order to run.

For example, let's say you have a script named `analyze.py` in your project directory along with a file `anaconda-project.yml`:

```
myproject/
   analyze.py
   anaconda-project.yml
```

The file `anaconda-project.yml` tells conda how to run your project:

```
commands:
  default:
    unix: "python analyze.py"
    windows: "python analyze.py"
```

There are separate command lines for Unix shells (Linux and macOS) and for Windows. You may target only one platform, and are not required to provide command lines for other platforms.

When you send your project to someone else, they can type `anaconda-project run` to run your script. The best part is that `anaconda-project run` makes sure that all prerequisites are set up *before* it runs the script.

Let's say your script requires a certain conda package to be installed. Add the `redis-py` package to `anaconda-project.yml` as a dependency using either the `packages` or `dependencies` key:

```
packages:
  - redis-py
```

Now when someone runs `anaconda-project run` the script is automatically run in a conda environment that has `redis-py` installed.

Here's another example. Let's say your script requires a huge data file that you don't want to put in source control and you don't want to email. You can add a requirement that the file will be downloaded locally:

```
downloads:
  MYDATAFILE:
    url: http://example.com/bigdatafile
    sha1: da39a3ee5e6b4b0d3255bfef95601890afd80709
```

Now when someone runs `anaconda-project run`, the file is downloaded if it hasn't been downloaded already, and the environment variable `MYDATAFILE` is set to the local filename of the data. In your `analyze.py` file you can write something like this:

```
import os
filename = os.getenv('MYDATAFILE')
if filename is None:
  raise Exception("Please use 'anaconda-project run' to start this script")
with open(filename, 'r') as input:
  data = input.read()
  # and so on
```

`anaconda-project` supports many other requirements, too. Instead of writing long documentation about how to set up your script before others can run it, simply put the requirements in a `anaconda-project.yml` file and let `anaconda-project` check and execute the setup automatically.

Multiple Commands

An `anaconda-project.yml` can list multiple commands. Each command has a name, and `anaconda-project run COMMAND_NAME` runs the command named `COMMAND_NAME`.

`anaconda-project list-commands` lists commands, along with a description of each command. To customize a command's description, add a `description:` field in `anaconda-project.yml`, like this:

```
commands:
  mycommand:
    unix: "python analyze.py"
    windows: "python analyze.py"
    description: "This command runs the analysis"
```

Special command types

Bokeh apps and notebooks have a shorthand syntax:

```
commands:
  foo:
    bokeh_app: foo
    description: "Runs the bokeh app in the foo subdirectory"
  bar:
    notebook: bar.ipynb
    description: "Opens the notebook bar.ipynb"
```

Notebook-specific options

Notebook commands can annotate that they contain a function registered with Anaconda Fusion:

```
commands:
  bar:
    notebook: bar.ipynb
    description: "Notebook exporting an Anaconda Fusion function."
    registers_fusion_function: true
```

If your notebook contains `@fusion.register` when you `anaconda-project init` or `anaconda-project add-command`, `registers_fusion_function: true` will be added automatically.

HTTP Commands

`anaconda-project` can be used to pack up web applications and run them on a server. Web applications include Bokeh applications, notebooks, APIs, and anything else that communicates with HTTP.

To make an `anaconda-project` command into a deployable web application, it has to support a list of command-line options.

Any command with `notebook:` or `bokeh_app:` automatically supports these options, because `anaconda-project` translates them into the native options supplied by the Bokeh and Jupyter command lines.

Shell commands (those with `unix:` or `windows:`) must support the semantics of these command-line options appropriately. They do *not* have to support the exact command line syntax used by `anaconda project run` as shell commands support jinja2 templating. For instance:

```
commands:
  myapp:
    unix: launch_flask_app.py --port {{port}} --host {{host}} --address {{address}}
    description: "Launches a Flask web app"
```

Here, `{{port}}`, `{{host}}` and `{{address}}` are jinja2 variables that are templated into the `--port`, `--host` and `--address` arguments of a hypothetical `launch_flask_app.py` script. These arguments are

just a few of the variables made available from the `--anaconda-project-` flags you can use when executing `anaconda-project run`:

- `--anaconda-project-host=HOST:PORT` can be specified multiple times and indicates a permitted value for the HTTP Host header. The value may include a port as well. There will be one `--anaconda-project-host` option for each host that browsers can connect to. This option specifies the application's public hostname:port and does not affect the address or port the application listens on. The last host specified is made available as the `host` jinja2 variable while the full list of hosts is available as the `hosts` variable.
- `--anaconda-project-port=PORT` indicates the local port the application should listen on; unlike the port which may be included in the `--anaconda-project-host` option, this port will not always be the one that browsers connect to. In a typical deployment, applications listen on a local-only port while a reverse proxy such as nginx listens on a public port and forwards traffic to the local port. In this scenario, the public port is part of `--anaconda-project-host` and the local port is provided as `--anaconda-project-port`. This setting is available for templating as the `port` jinja2 variable.}
- `--anaconda-project-address=IP` indicates the IP address the application should listen on. Unlike the host which may be included in the `--anaconda-project-host` option, this address may not be the one that browsers connect to. This setting is available for templating as the `address` jinja2 variable.
- `--anaconda-project-url-prefix=PREFIX` gives a path prefix that should be the first part of the paths to all routes in your application. For example, if you usually have a page `/foo.html`, and the prefix is `/bar`, you would now have a page `/bar/foo.html`. This setting is available for templating as the `url_prefix` jinja2 variable.
- `--anaconda-project-no-browser` means "don't open a web browser when the command is run." If your command never opens a web browser anyway, you should accept but ignore this option. This setting is available for templating as the `no_browser` jinja2 variable. When this switch is present, the value of `no_browser` is `True`.
- `--anaconda-project-iframe-hosts=HOST:PORT` gives a value to be included in the `Content-Security-Policy` header as a value for `frame-ancestors` when you serve an HTTP response. The effect of this is to allow the page to be embedded in an iframe by the supplied HOST:PORT. This setting is available for templating as the `iframe-hosts` jinja2 variable.
- `--anaconda-project-use-xheaders` tells your application that it's behind a reverse proxy and can trust "X-" headers, such as `X-Forwarded-For` or `X-Host`. This setting is available for templating as the `use_xheaders` jinja2 variable. When this switch is present, the value of `use_xheaders` is `True`.

As an alternative to the templating approach, you may choose to write `launch_flask_app.py` in such a way that it supports the above command line flags and switches directly. In this case, you need to specify `supports_http_options: true`:

```
commands:
  myapp:
    unix: {{PROJECT_DIR}}/launch_flask_app.py
    supports_http_options: true
    description: "Launches a Flask web app"
```

This example illustrates that in addition to the jinja2 variables described above, all environment variables are also available for templating, including `PROJECT_DIR` and conda related environment variables such as `CONDA_PREFIX` and `CONDA_DEFAULT_ENV`.

Environments and Channels

You can configure packages in a top level `packages` or `dependencies` section of the `anaconda-project.yml` file, as we discussed earlier:

```
packages:
  - redis-py
```

You can also add specific conda channels to be searched for packages:

```
channels:
  - conda-forge
```

`anaconda-project` creates an environment in `envs/default` by default. But if you prefer, you can have multiple named environments available in the `envs` directory. To do that, specify an `env_specs:` section of your `anaconda-project.yml` file:

```
env_specs:
  default:
    packages:
      - foo
      - bar
    channels:
      - conda-forge
  python27:
    description: "Uses Python 2 instead of 3"
    packages:
      - python < 3
    channels:
      - https://example.com/somechannel
```

An environment specification or "env spec" is a description of an environment, describing the packages that the project requires to run. By default, env specs are instantiated as actual Conda environments in the `envs` directory inside your project.

In the above example we create two env specs, which will be instantiated as two environments, `envs/default` and `envs/python27`.

To run a project using a specific env spec, use the `--env-spec` option:

```
anaconda-project run --env-spec myenvname
```

If you have top level `channels` or `packages` sections in your `anaconda-project.yml` file (not in the `env_specs:` section), those channels and packages are added to all env specs.

The default env spec can be specified for each command, like this:

```
commands:
  mycommand:
    unix: "python ${PROJECT_DIR}/analyze.py"
    windows: "python %PROJECT_DIR%\analyze.py"
    env_spec: my_env_spec_name
```

Env specs can also inherit from one another. List a single env spec or a list of env specs to inherit from, something like this:

```
env_specs:
  test_packages:
    description: "Packages used for testing"
    packages:
      - pytest
      - pytest-cov
  app_dependencies:
```

(continues on next page)

(continued from previous page)

```
    description: "Packages used by my app"
    packages:
      - bokeh
  app_test_dependencies:
    description: "Packages used to test my app"
    inherit_from: [test_packages, app_dependencies]

commands:
  default:
     unix: start_my_app.py
     env_spec: app_dependencies
  test:
     unix: python -m pytest myapp/tests
     env_spec: app_test_dependencies
```

pip packages

Underneath any *packages:* or *dependencies:* section, you can add a *pip:* section with a list of pip requirement specifiers.

```
packages:
   - condapackage1
   - pip:
     - pippackage1
     - pippackage2
```

Locking package versions

Any env spec can be "locked", which means it specifies exact versions of all packages to be installed, kept in `anaconda-project-lock.yml`.

Hand-creating `anaconda-project-lock.yml` isn't recommended. Instead, create it with the `anaconda-project lock` command, and update the versions in the configuration file with `anaconda-project update`.

Locked versions are distinct from the "logical" versions in `anaconda-project.yml`. For example, your `anaconda-project.yml` might list that you require `bokeh=0.12`. The `anaconda-project lock` command expands that to an *exact* version of Bokeh such as `bokeh=0.12.4=py27_0`. It will also list exact versions of all Bokeh's dependencies transitively, so you'll have a longer list of packages in `anaconda-project-lock.yml`. For example:

```
locking_enabled: true

env_specs:
  default:
    locked: true
    env_spec_hash: eb23ad7bd050fb6383fcb71958ff03db074b0525
    platforms:
    - linux-64
    - win-64
    packages:
      all:
      - backports=1.0=py27_0
      - backports_abc=0.5=py27_0
```

(continues on next page)

(continued from previous page)

```
    - bokeh=0.12.4=py27_0
    - futures=3.0.5=py27_0
    - jinja2=2.9.5=py27_0
    - markupsafe=0.23=py27_2
    - mkl=2017.0.1=0
    - numpy=1.12.1=py27_0
    - pandas=0.19.2=np112py27_1
    - pip=9.0.1=py27_1
    - python-dateutil=2.6.0=py27_0
    - python=2.7.13=0
    - pytz=2016.10=py27_0
    - pyyaml=3.12=py27_0
    - requests=2.13.0=py27_0
    - singledispatch=3.4.0.3=py27_0
    - six=1.10.0=py27_0
    - ssl_match_hostname=3.4.0.2=py27_1
    - tornado=4.4.2=py27_0
    - wheel=0.29.0=py27_0
    unix:
    - openssl=1.0.2k=1
    - readline=6.2=2
    - setuptools=27.2.0=py27_0
    - sqlite=3.13.0=0
    - tk=8.5.18=0
    - yaml=0.1.6=0
    - zlib=1.2.8=3
    win:
    - setuptools=27.2.0=py27_1
    - vs2008_runtime=9.00.30729.5054=0
```

By locking your versions, you can make your project more portable. When you share it with someone else or deploy it on a server or try to use it yourself in a few months, you'll get the same package versions you've already used for testing. If you don't lock your versions, you may find that your project stops working due to changes in its dependencies.

When you're ready to test the latest versions of your dependencies, run `anaconda-project update` to update the versions in `anaconda-project-lock.yml` to the latest available.

If you check `anaconda-project-lock.yml` into revision control (such as git), then when you check out old versions of your project you'll also get the dependencies those versions were tested with. And you'll be able to see changes in your dependencies over time in your revision control history.

Additionally, all pip packages added to the `anaconda-project.yml` file or installed as dependencies will be added to the `anaconda-project-lock.yml` file similar to the output of `pip freeze`. For example, see the following `anaconda-project-lock.yml` file that matches the following package specification.

```
packages:
  - python=3.8
  - pip:
    - requests
```

```
locking_enabled: true

env_specs:
  default:
    locked: true
    env_spec_hash: 292a009a194f1ca1d3432c824df6ff51a7aef388
```

(continues on next page)

(continued from previous page)

```
  platforms:
  - linux-64
  - osx-64
  - win-64
  packages:
    all:
    - wheel=0.36.2=pyhd3eb1b0_0
    linux-64:
    - _libgcc_mutex=0.1=main
    - ca-certificates=2021.4.13=h06a4308_1
    - certifi=2020.12.5=py38h06a4308_0
    - ld_impl_linux-64=2.33.1=h53a641e_7
    - libffi=3.3=he6710b0_2
    - libgcc-ng=9.1.0=hdf63c60_0
    - libstdcxx-ng=9.1.0=hdf63c60_0
    - ncurses=6.2=he6710b0_1
    - openssl=1.1.1k=h27cfd23_0
    - pip=21.0.1=py38h06a4308_0
    - python=3.8.8=hdb3f193_5
    - readline=8.1=h27cfd23_0
    - setuptools=52.0.0=py38h06a4308_0
    - sqlite=3.35.4=hdfb4753_0
    - tk=8.6.10=hbc83047_0
    - xz=5.2.5=h7b6447c_0
    - zlib=1.2.11=h7b6447c_3
    osx-64:
    - ca-certificates=2021.4.13=hecd8cb5_1
    - certifi=2020.12.5=py38hecd8cb5_0
    - libcxx=10.0.0=1
    - libffi=3.3=hb1e8313_2
    - ncurses=6.2=h0a44026_1
    - openssl=1.1.1k=h9ed2024_0
    - pip=21.0.1=py38hecd8cb5_0
    - python=3.8.8=h88f2d9e_5
    - readline=8.1=h9ed2024_0
    - setuptools=52.0.0=py38hecd8cb5_0
    - sqlite=3.35.4=hce871da_0
    - tk=8.6.10=hb0a8c7a_0
    - xz=5.2.5=h1de35cc_0
    - zlib=1.2.11=h1de35cc_3
    win-64:
    - ca-certificates=2021.4.13=haa95532_1
    - certifi=2020.12.5=py38haa95532_0
    - openssl=1.1.1k=h2bbff1b_0
    - pip=21.0.1=py38haa95532_0
    - python=3.8.8=hdbf39b2_5
    - setuptools=52.0.0=py38haa95532_0
    - sqlite=3.35.4=h2bbff1b_0
    - vc=14.2=h21ff451_1
    - vs2015_runtime=14.27.29016=h5e58377_2
    - wincertstore=0.2=py38_0
    pip:
    - chardet==4.0.0
    - idna==2.10
    - requests==2.25.1
    - urllib3==1.26.4
```

Specifying supported platforms

Whenever you lock or update a project, dependencies are resolved for all platforms that the project supports. This allows you to do your work on Windows and deploy to Linux, for example.

`anaconda-project lock` by default adds a `platforms: [linux-64,osx-64,win-64]` line to `anaconda-project.yml`. If you don't need to support these three platforms, or want different ones, change this line. Updates will be faster if you support fewer platforms. Also, some projects only work on certain platforms.

The `platforms:` line does nothing when a project is unlocked.

Platform names are the same ones used by `conda`. Possible values in `platforms:` include `linux-64`, `linux-32`, `win-64`, `win-32`, `osx-64`, `osx-32`, `linux-armv6l`, `linux-armv7l`, `linux-ppc64le`, and so on.

In `anaconda-project.yml` a `platforms:` list at the root of the file will be inherited by all env specs, and then each env spec can add (but not subtract) additional platforms. It works the same way as the `channels:` list in this respect. `inherit_from:` will also cause platforms to be inherited.

Enabling and disabling locked versions

If you delete `anaconda-project-lock.yml`, the project will become "unlocked."

If you have an `anaconda-project-lock.yml`, the `locking_enabled:` field indicates whether env specs are locked by default. Individual env spec sections in `anaconda-project-lock.yml` can then specify `locked: true` or `locked: false` to override the default on a per-env-spec basis.

`anaconda-project unlock` turns off locking for all env specs and `anaconda-project lock` turns on locking for all env specs.

Updating locked versions after editing an env spec

If you use commands such as `anaconda-project add-packages` or `anaconda-project add-env-spec` to edit your `anaconda-project.yml`, then `anaconda-project-lock.yml` will automatically be kept updated.

However, if you edit `anaconda-project.yml` by hand and change an env spec, you'll need to run `anaconda-project update` to update `anaconda-project-lock.yml` to match.

If locking isn't enabled for the project or for the env spec, there's no need to `anaconda-project update` after editing your env spec.

Requiring environment variables to be set

Anything in the `variables:` section of a `anaconda-project.yml` file is considered an environment variable needed by your project. When someone runs your project, `anaconda-project` asks them to set these variables.

For example:

```
variables:
  - AMAZON_EC2_USERNAME
  - AMAZON_EC2_PASSWORD
```

Now in your script, you can use `os.getenv()` to get these variables.

NOTE: This is a much better option than hardcoding passwords into your script, which can be a security risk.

Variables that contain credentials

Variables that end in `_PASSWORD`, `_ENCRYPTED`, `_SECRET_KEY`, or `_SECRET` are treated sensitively by default. This means that if `anaconda-project` stores a value for them in `anaconda-project.yml` or `anaconda-project-local.yml` or elsewhere, that value is encrypted. NOTE: `anaconda-project-local.yml` stores and encrypts the value that you enter when prompted.

To force a variable to be encrypted or not encrypted, add the `encrypted` option to it in `anaconda-project.yml`, like this:

```
variables:
  # let's encrypt the password but not the username
  AMAZON_EC2_USERNAME: { encrypted: false }
  AMAZON_EC2_PASSWORD: { encrypted: true }
```

NOTE: The value of the environment variable is NOT encrypted when passed to your script; the encryption happens only when we save the value to a config file.

Variables with default values

If you make the `variables:` section a dictionary instead of a list, you can give your variables default values. Anything in the environment or in `anaconda-project-local.yml` overrides these defaults. To omit a default for a variable, set its value to either `null` or `{}`.

For example:

```
variables:
  ALPHA: "default_value_of_alpha"
  BRAVO: null # no default for BRAVO
  CHARLIE: {} # no default for CHARLIE
  # default as part of options dict, needed if you also
  # want to set some options such as 'encrypted: true'
  DELTA: { default: "default_value_of_delta" }
  ECHO: { default: "default_value_of_echo", encrypted: true }
```

Variables can have custom description strings

A variable can have a 'description' field, which will be used in UIs which display the variable.

For example:

```
variables:
  SALES_DB_PASSWORD: {
     description: "The password for the sales database. Ask jim@example.com if you don
↪'t have one."
  }
```

Variables that are always set

`anaconda-project` ensures that the following variables are always set:

- `PROJECT_DIR` is set to the top level directory of your project
- `CONDA_ENV_PATH` is set to the filesystem location of the current conda environment
- `PATH` includes the binary directory from the current conda environment

These variables always exist and can always be used in your Python code. For example, to get a file from your project directory, try this in your Python code (notebook or script):

```
import os
project_dir = os.getenv("PROJECT_DIR")
my_file = os.path.join(project_dir, "my/file.txt")
```

Services

TIP: Services are a proof-of-concept demo feature for now.

Services can be automatically started, and their address can be provided to your code by using an environment variable.

For example, you can add a services section to your `anaconda-project.yml` file:

```
services:
  REDIS_URL: redis
```

Now when someone else runs your project, `anaconda-project` offers to start a local instance of `redis-server` automatically.

There is also a long form of the above service configuration:

```
services:
  REDIS_URL: { type: redis }
```

and you can set a default and any options a service may have:

```
services:
  REDIS_URL:
     type: redis
     default: "redis://localhost:5895"
```

Right now there is only one supported service (Redis) as a demo. We expect to support more soon.

File Downloads

The `downloads:` section of the `anaconda-project.yml` file lets you define environment variables that point to downloaded files. For example:

```
downloads:
  MYDATAFILE:
    url: http://example.com/bigdatafile
    sha1: da39a3ee5e6b4b0d3255bfef95601890afd80709
```

Rather than *sha1*, you can use whatever integrity hash you have; supported hashes are `md5`, `sha1`, `sha224`, `sha256`, `sha384`, `sha512`.

NOTE: The download is checked for integrity ONLY if you specify a hash.

You can also specify a filename to download to, relative to your project directory. For example:

```
downloads:
  MYDATAFILE:
    url: http://example.com/bigdatafile
    filename: myfile.csv
```

This downloads to `myfile.csv`, so if your project is in `/home/mystuff/foo` and the download succeeds, `MYDATAFILE` is set to `/home/mystuff/foo/myfile.csv`.

If you do not specify a filename, `anaconda-project` picks a reasonable default based on the URL.

To avoid the automated download, it's also possible for someone to run your project with an existing file path in the environment. On Linux or Mac, that looks like:

```
MYDATAFILE=/my/already/downloaded/file.csv anaconda-project run
```

Conda can auto-unzip a zip file as it is downloaded. This is the default if the URL path ends in ".zip" unless the filename also ends in ".zip". For URLs that do not end in ".zip", or to change the default, you can specify the "unzip" flag:

```
downloads:
  MYDATAFILE:
    url: http://example.com/bigdatafile
    unzip: true
```

The `filename` is used as a directory and the zip file is unpacked into the same directory, unless the zip contains a single file or directory with the same name as `filename`. In that case, then the two are consolidated.

EXAMPLE: If your zip file contains a single directory `foo` with file `bar` inside that, and you specify downloading to filename `foo`, then you'll get `PROJECT_DIR/foo/bar`, not `PROJECT_DIR/foo/foo/bar`.

Describing the Project

By default, `anaconda-project` names your project with the same name as the directory in which it is located. You can give it a different name in `anaconda-project.yml`:

```
name: myproject
```

You can also have an icon file, relative to the project directory:

```
icon: images/myicon.png
```

No need to edit `anaconda-project.yml` directly

You can edit `anaconda-project.yml` with the `anaconda-project` command.

To add a download to `anaconda-project.yml`:

```
anaconda-project add-download MYFILE http://example.com/myfile
```

To add a package:

```
anaconda-project add-packages redis-py
```

To ask for a running Redis instance:

```
anaconda-project add-service redis
```

3.4 Help and support

To ask questions or submit bug reports, use the Github Issue Tracker.

3.4.1 Paid support

Anaconda Project is an open source project that originated at Anaconda, Inc. Continuum offers paid training and support.

3.4.2 Send feedback

Help us make this documentation better. Send feedback about the Project documentation to documentation@continuum.io.