Contents

[**Conda Essentials** 3](#_Toc531537751)

[1. Installing Packages 3](#_Toc531537752)

[1.1 What are packages and why are they needed? 3](#_Toc531537753)

[1.2 What version of conda do I have? 4](#_Toc531537754)

[1.3 Install a conda package (I) 5](#_Toc531537755)

[1.4 Install a conda package (II) 6](#_Toc531537756)

[1.5 What is semantic versioning? 7](#_Toc531537757)

[1.6 Which package version is installed? 8](#_Toc531537758)

[1.7 Install a specific version of a package (I) 9](#_Toc531537759)

[1.8 Install a specific version of a package (II) 10](#_Toc531537760)

[1.9 Update a conda package 11](#_Toc531537761)

[1.10 Remove a conda package 12](#_Toc531537762)

[1.11 Search for available package versions? 12](#_Toc531537763)

[1.12 Find dependencies for a package versions? 13](#_Toc531537764)

[2. Utilizing Channels 14](#_Toc531537765)

[2.1 Channels and why are they needed? 14](#_Toc531537766)

[2.2 Searching within channels 15](#_Toc531537767)

[2.3 Searching across channels 16](#_Toc531537768)

[2.4 Default, non-default, and special channels 17](#_Toc531537769)

[2.5 Installing from a channel 18](#_Toc531537770)

[3. Working with Environments 19](#_Toc531537771)

[3.1 Environments and why are they needed? 19](#_Toc531537772)

[3.2 Which environment am I using? 20](#_Toc531537773)

[3.3 What packages are installed in an environment? (I) 21](#_Toc531537774)

[3.4 What packages are installed in an environment? (II) 22](#_Toc531537775)

[3.5 Switch between environments 23](#_Toc531537776)

[3.6 Remove an environment 24](#_Toc531537777)

[3.7 Create a new environment 25](#_Toc531537778)

[3.8 Export an environment 26](#_Toc531537779)

[3.9 Create an environment from a shared specification 27](#_Toc531537780)

[4. Case Study on Using Environments 28](#_Toc531537781)

[4.1 Compatibility with different versions 28](#_Toc531537782)

[4.2 Updating a script 29](#_Toc531537783)

**Conda Essentials**

# Installing Packages

## What are packages and why are they needed?

Conda packages are files containing a bundle of resources: usually libraries and executables, but not always. In principle, Conda packages can include data, images, notebooks, or other assets. The command-line tool conda is used to install, remove and examine packages; other tools such as the GUI *Anaconda Navigator* also expose the same capabilities. This course focuses on the conda tool itself (you'll see use cases other than package management in later chapters).

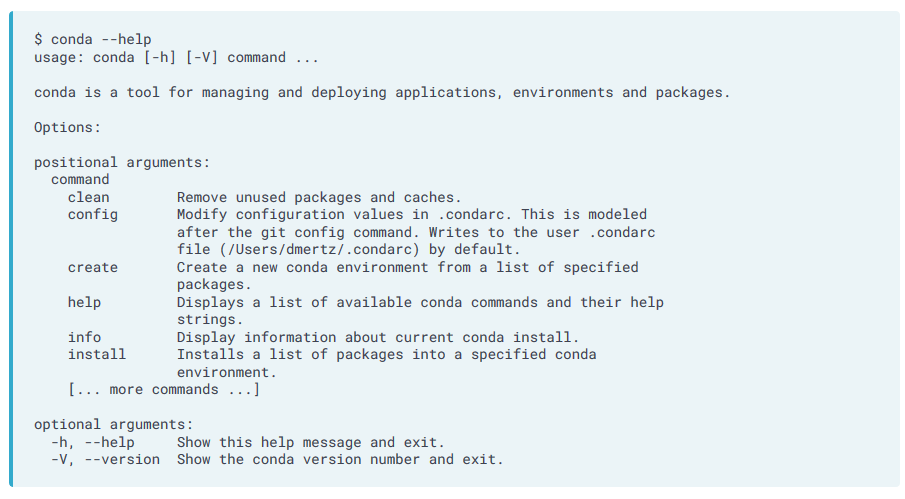
Conda packages are most widely used with Python, but that's not all. Nothing about the Conda package format or the conda tool itself assumes any specific programming language. Conda packages can also be used for bundling libraries in other languages (like R, Scala, Julia, etc.) or simply for distributing pure binary executables generated from *any* programming language.

One of the powerful aspects of conda—both the tool and the package format—is that dependencies are taken care of. That is, when you install any Conda package, any other packages needed get installed automatically. Tracking and determining software dependencies is a hard problem that package managers like Conda are designed to solve.

A Conda package, then, is a file containing all files needed to make a given program execute correctly on a given system. Moreover, a Conda package can contain binary artifacts specific to a particular platform or operating system. Most packages (and their dependendencies) are available for Windows (win-32 or win-64), for OSX (osx-64), and for Linux (linux-32 or linux-64). A small number of Conda packages are available for more specialized platforms (e.g., Raspberry Pi 2 or POWER8 LE). As a user, you do not need to specify the platform since Conda will simply choose the Conda package appropriate for the platform you are using.

## What version of conda do I have?

The tool conda takes a variety of commands and arguments. Most of the time, you will use conda COMMAND OPTIONS --SWITCH. You will learn the collection of COMMANDs available in the next lessons. A summary is available on the help screen:



Run a command to determine what version of conda you have installed.

**Hint**

Run conda with --version.

## Install a conda package (I)

Just as you can get help on conda as a whole, you can get help on commands within it. You will often use the command conda install; you can look at the corresponding help documentation using the terminal window. That is, run conda install --help and read through the output.

How is the positional argument package\_spec defined in the documentation for conda install?

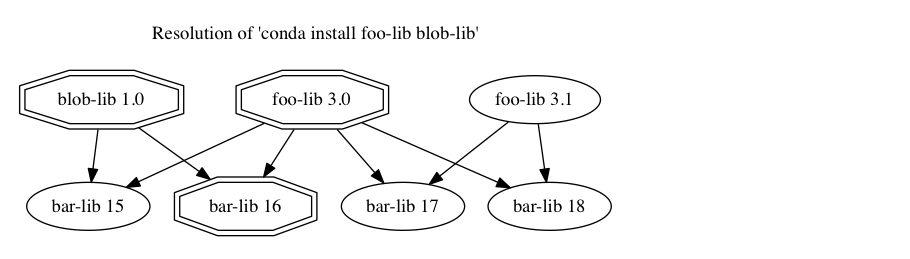
## Install a conda package (II)

Installing a package is largely a matter of listing the name(s) of packages to install after the command conda install. But there is more to it behind the scenes. The versions of packages to install (along with all their dependencies) must be compatible with all versions of other software currently installed. Often this "satisfiability" constraint depends on choosing a package version compatible with a particular version of Python that is installed. Conda is special among "package managers" in that it always guarantees this consistency; you will see the phrase "Solving environment..." during installation to indicate this computation.

For example, you may simply instruct conda to install foo-lib. The tool first determines which operating system you are running, and then narrows the match to candidates made for this platform. Then, conda determines the version of Python on the system (say 3.7), and chooses the package version for -py37. But, beyond those simple limits, all dependencies are checked.

Suppose foo-lib is available in versions 1.0, 1.1, 1.2, 2.0, 2.1, 2.2, 2.3, 3.0, 3.1 (for your platform and Python version). As a first goal, conda attempts to choose the latest version of foo-lib. However, maybe foo-lib depends on bar-lib, which itself is available in various versions (say 1 through 20 in its versioning scheme). It might be that foo-lib 3.1 is compatible with bar-lib versions 17, 18, and 19; but blob-lib (which is already installed) is compatible only with versions of bar-lib less than 17. Therefore, conda would examine the compatibility of foo-lib 3.0 as a fallback. In this hypothetical, foo-lib 3.0 is compatible with bar-lib 16, so that version is chosen (bar-lib is also updated to the latest compatible version 16 in the same command if an earlier version is currently installed).

Visually (octagons mark chosen versions):



Install the package cytoolz using conda.

**Hint**

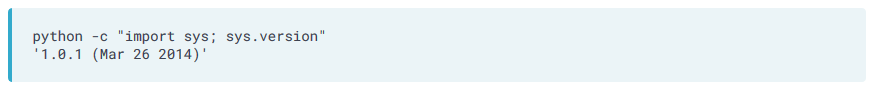
Run conda with the install command and then specify the package name.

## What is semantic versioning?

Most Conda packages use a system called [*semantic versioning*](https://semver.org) to identify distinct versions of a software package unambiguously. Version labels are usually chosen by the project authors, not necessarily the same people who bundle the project as a Conda package. There is no technical requirement that a project author's version label coincides with a Conda package version label, but the convention of doing so is almost always followed. Semantic version labels can be compared lexicographically and make it easy to determine which of two versions is the later version.

Under [semantic versioning](https://semver.org), software is labeled with a three-part version identifier of the form MAJOR.MINOR.PATCH; the label components are non-negative integers separated by periods. Assuming all software starts at version 0.0.0, the MAJOR version number is increased when significant new functionality is introduced (often with corresponding API changes). Increases in the MINOR version number generally reflect improvements (e.g., new features) that avoid backward-incompatible API changes. For instance, adding an optional argument to a function API (in a way that allows old code to run unchanged) is a change worthy of increasing the MINOR version number. An increment to the PATCH version number is approriate mostly for bug fixes that preserve the same MAJOR and MINOR revision numbers. Software patches do not typically introduce new features or change APIs at all (except sometimes to address security issues).

Many command-line tools display their version identifier by running tool --version. This information can sometimes be displayed or documented in other ways. For example, suppose on some system, a certain version of Python is installed, and you inquire about it like this:

Looking at the output above, which statement below accurately characterizes the semantic versioning of this installed Python?

## Which package version is installed?

Because conda installs packages automatically, it's hard to know which package versions are actually on your system. That is, packages you didn't install explicitly get installed for you to resolve another package's dependencies.

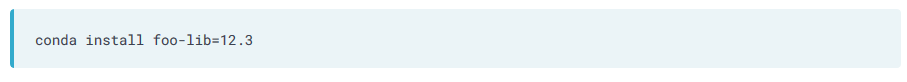
Fortunately, the command conda list comes to your aid to query the current state. By itself, this lists all packages currently installed.  
You can use conda list --help to see how to extract specific packages from that list or how to display it in different formats.

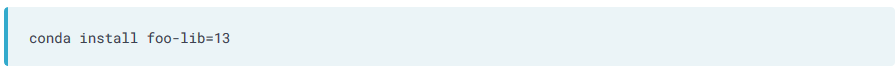
Select the exact version of the package attrs installed in the current session.

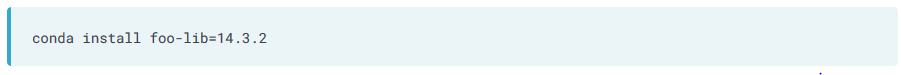
## Install a specific version of a package (I)

Sometimes there are reasons why you need to use a specific version of a package, not necessarily simply the latest version compatible with your other installed software. You may have scripts written that depend on particular older APIs, or you may have received code written by colleagues who used specific versions and you want to guarantee replication of the same behavior. Likewise, you may be writing code that you intend to pass to other users who you know to be using specific package versions on their systems (perhaps as a company standard, for example).

conda allows you to install software versions in several flexible ways. Your most common pattern will probably be prefix notation, using semantic versioning. For example, you might want a MAJOR and MINOR version, but want conda to select the most up-to-date PATCH version within that series. You could spell that as:

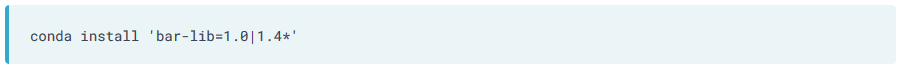
Or similarly, you may want a particular major version, and prefer conda to select the latest compatible MINOR version as well as PATCH level. You could spell that as:

If you want to narrow the installation down to an exact PATCH level, you can specify that as well with:

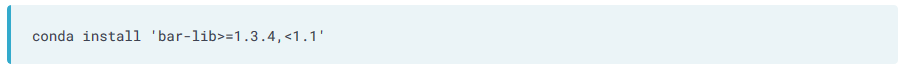
Keep in mind that relaxing constraints may allow for satisfying multiple dependencies among installed software. Occasionally you will try to install some software version that is simply inconsistent with other software installed, and conda will warn you about that rather than install anything.

## Install a specific version of a package (II)

Most commonly, you'll use prefix-notation to specify the package version(s) to install. But conda offers even more powerful comparison operations to narrow versions. For example, if you wish to install either bar-lib versions 1.0, 1.4 or 1.4.1b2, but definitely not version 1.1, 1.2 or 1.3, you could use:

This may seem odd, but you might know, for example, that a bug was introduced in 1.1 that wasn't fixed until 1.4. You would prefer the 1.4 series, but, if it is incompatible with other packages, you can settle for 1.0. Notice we have used single quotes around the version expression in this case because several of the symbols in these more complex patterns have special meanings in terminal shells. It is easiest just to quote them.

With conda you can also use inequality comparisons to select candidate versions (still resolving dependency consistency). Maybe the bug above was fixed in 1.3.5, and you would like either the latest version available (perhaps even 1.5 or 2.0 have come out), but still avoiding versions 1.1 through 1.3.4. You could spell that as:

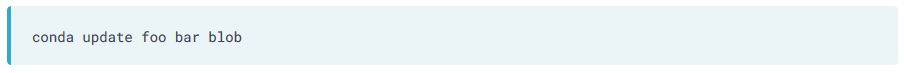
For this exercise, install the latest compatible version of attrs that is later than version 16, but earlier than version 17.3. Which version gets installed?

## Update a conda package

Closely related to installing a particular version of a conda package is updating the installed version to the latest version possible that remains compatible with other installed software. conda will determine if it is possible to update dependencies of the package(s) you are directly updating, and do so if resolvable. At times, the single specified package will be updated exclusively since the current dependencies are correct for the new version. Obviously, at other times updating will do nothing because you are already at the latest version possible.

The command conda update PKGNAME is used to perform updates. Update is somewhat less "aggressive" than install in the sense that installing a specific (later) version will revise the versions in the dependency tree to a greater extent than an update. Often update will simply choose a later PATCH version even though potentially a later MAJOR or MINOR version could be made compatible with other installed packages.

Note that this conda command, as well as most others allow specification of multiple packages on the same line. For example, you might use:

To bring all of foo, bar, and blob up to the latest compatible versions mutually satisfiable.

## Remove a conda package

Finally, in direct package management, sometimes you want to remove a package. This is straightforward using the command conda remove PKGNAME. As with other commands, you may also optionally specify multiple packages separated by spaces.

Note that conda always tries to use the most recent versions of installed software that are compatible. Therefore, sometimes removing one package allows another package to be upgraded implicitly because only the removed package was requiring the older version of the dependency.

## Search for available package versions?

Sometimes you want to see what versions of a package are available as conda packages. By default conda search looks for those matching your platform (although switches allow tweaking this behavior).

## Find dependencies for a package versions?

The conda info command reports a variety of details about a specific package. The syntax for specifying just one version is a little bit complex, but prefix notation is allowed here (just as you would with conda install).

For example, running conda info cytoolz=0.8.2 will report on all available package versions. As this package has been built for a variety of Python versions, a number of packages will be reported on. You can narrow your query further with, e.g.:

You may use the \* wildcard within the match pattern. This is often useful to match 'foo=1.2.3=py36\*' because recent builds have attached the hash of the build at the end of the Python version string, making the exact match unpredicatable.

Determine the dependencies of the package numpy 1.13.1 with Python 3.6.0 on your current platform.

###### Hint

Run conda info with specifiers for numpy version and Python version.

# Utilizing Channels

## Channels and why are they needed?

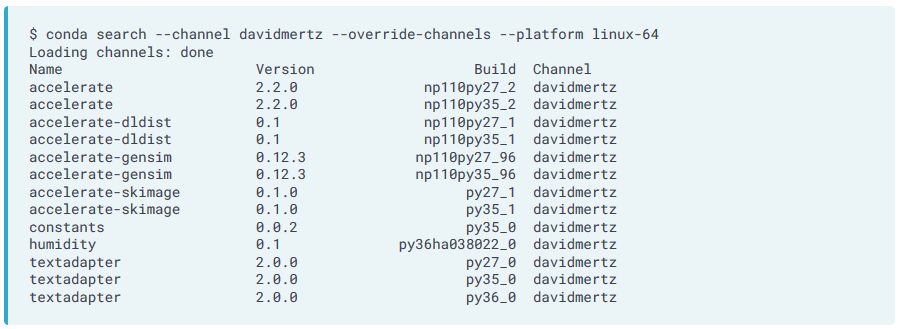
All Conda packages we've seen so far were published on the main or default channel of Anaconda Cloud. A Conda channel is an identifier of a path (e.g., as in a web address) from which Conda packages can be obtained. Using the public cloud, installing without specifying a channel points to the main channel at <https://repo.continuum.io/pkgs/main>; where hundreds of packages are available. Although covering a wide swath, the main channel contains only packages that are (moderately) curated by Anaconda Inc. Given finite resources and a particular area focus, not all genuinely worthwhile packages are vetted by Anaconda Inc.

If you happen to be working in a firewalled or airgapped environment with a private installation of Anaconda Repository, your default channel may point to a different (internal) URL, but the same concepts will apply.

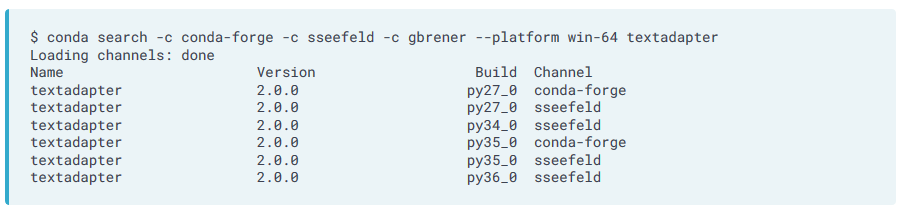
Anyone may register for an account with Anaconda Cloud, thereby creating their own personal Conda channel. This is covered in the companion course Conda for Building & Distributing Packages (along with creating and uploading your own packages). For this course, just understand that many users have accounts and corresponding channels.

## Searching within channels

If a particular colleague or other recognized user may have published a package useful to you, you can search for it using the anaconda search command. For example, David Mertz, the principal author of this course, has a channel and Anaconda Cloud account called davidmertz. You can search his channel using the command below; the option --channel (or -c for short) specifies the channel to search. Particular users may have published more niche software you would like to use; for example, colleagues of yours may publish packages of special use in your field or topic area.

In this case, the switch --override-channels is used to prevent searching on default channels. The switch --platform is used to select a platform that may differ from the one on which the search is run (absent the switch, the current computer's platform is used).

The first search is unusual in that it does not specify a package name, which is more typical actual use. For example, you might want to know which versions of the package of textadapter for the win-64 platform are available for any version of Python (assuming you know in which channels to look):

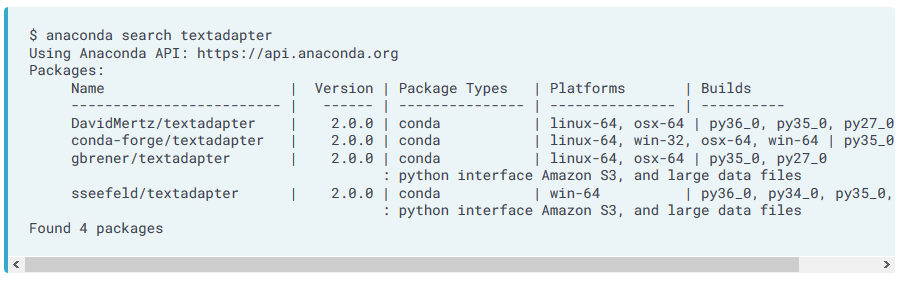


Based on the examples shown, in which of the channels used in the examples above could you find an osx-64 version of textadapter for Python 3.6?

Conda search –c conda-forge –c davidmertz –plattform osx-64 textadapter

## Searching across channels

Although the conda command and its subcommands are used for nearly everything in this course, the package anaconda-client provides the command anaconda that searches in a different manner that is often more useful. For instance, you may know the name of the textadapter package, but you may not know in which channel (or channels) it may be published (or by which users). You can search across all channels and all platforms using:



Following this example, use anaconda search to determine the latest available version of the package boltons.

## Default, non-default, and special channels

The default channel on Anaconda Cloud is curated by Anaconda Inc., but another channel called conda-forge also has a special status. This channel does not operate any differently than other channels, whether those others are associated with an individual or organization, but it acts as a kind of "community curation" of relatively well-vetted packages. The GitHub page for the conda-forge project at <https://github.com/conda-forge> describes it as: "A community led collection of recipes, build infrastructure and distributions for the conda package manager."

Apart from the somewhat more organized conda-forge channel/project, Anaconda Cloud channels are relatively anarchic. Much like GitHub repos or packages on the Python Package Index (PyPI), anyone is free to upload whatever projects they like to conda-forge (as long as they are assembled as Conda packages, that is, but this is a minor restriction).

You should generally trust or rely only on packages sourced from reputable channels. There is no inherent rating system for channels or their packages. However, you are likely to trust your colleagues, your organization, well-known people in your software or data science communities, and so on.

conda-forge is almost certainly the most widely used channel on Anaconda Cloud. In fact, it has very many more packages than the main channel itself. Use the tools from this course to make a rough estimate of the number of packages available on the conda-forge channel for the linux-64 platform (i.e., the platform used for this session).

###### Hint

It may be useful to use the tool wc (word count) on your query output. For example, if you wanted to know how many lines there are in Markdown (.md) files in the current directory, you could pipe to wc as follows (performed somewhere other than the current session for the example).

$ cat \*.md | wc -l

820

Or to ask how many lines, words, and characters there are on Project Gutenberg's copy of Marcel Proust's Swan's Way, you could run:

$ wget -qO- http://www.gutenberg.org/cache/epub/7178/pg7178.txt | wc

17461 198262 1122204

Query the entire channel and count the lines of output.

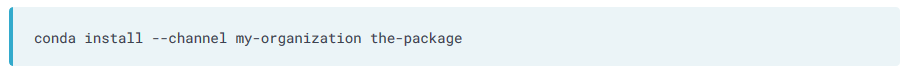
Conda search –c conda-forge –platform linux-64 | wc

## Installing from a channel

We saw in the last exercise that there are about 30,000 linux-64 packages on conda-forge. Across all the channels there are about 50,000 packages, most of those for at least 3 of of the 5 main platforms (osx-64, linux-32, linux-64, win-32, win-64; 32-bit support is of diminishing importance compared to 64-bit). There are around 2500 channels that have been active in the last 6 months; most are individual users, but a fair number belonging to projects or organizations. A majority of package names are published by more than one different channel; sometimes just as a copy, other times with a tweak or compiler optimization, or in a different version.

The whole point of having channels is to be able to install packages from them. For this exercise, you will install a version of a package not available on the default channel. Adding a channel to install from simply requires using the same --channel or -c switch we have seen in other conda commands, but with the conda install command.

For example:



A package named youtube-dl exists on conda-forge but is not available on the default channel. Please install it.

conda install --channel conda-forge youtube-dl

You should examine what software is installed in your current environment now. You should notice that unlike other packages, the newly install youtube-dl came from a non-default channel.

###### Hint

Use conda list to view installed packages.

# Working with Environments

## Environments and why are they needed?

Conda environments allow multiple incompatible versions of the same (software) package to coexist on your system. An environment is simply a filepath containing a collection of mutually compatible packages. By isolating distinct versions of a given package (and their dependencies) in distinct environments, those versions are all available to work on particular projects or tasks.

There are a large number of reasons why it is best practice to use environments, whether as a data scientist, software developer, or domain specialist. Without the concept of environments, users essentially rely on and are restricted to whichever particular package versions are installed globally (or in their own user accounts) on a particular machine. Even when one user moves scripts between machines (or shares them with a colleague), the configuration is often inconsistent in ways that interfere with seamless functionality. Conda environments solve both these problems. You can easily maintain and switch between as many environments as you like, and each one has exactly the collection of packages that you want.

For example, you may develop a project comprising scripts, notebooks, libraries, or other resources that depend on a particular collection of package versions. You later want to be able to switch flexibly to newer versions of those packages and to ensure the project continues to function properly before switching wholly. Or likewise, you may want to share code with colleagues who are required to use certain package versions. In this context, an environment is a way of documenting a known set of packages that correctly support your project.

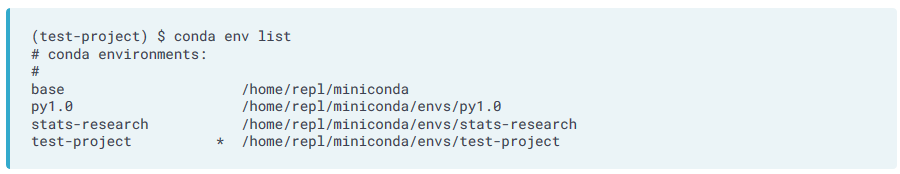
## Which environment am I using?

When using conda, you are always in some environment, but it may be the default (called the base or root environment). Your current environment has a name and contains a collection of packages currently associated with that environment. There are a few ways to determine the current environment.

Most obviously, at a terminal prompt, the name of the current environment is usually prepended to the rest of your prompt in parentheses. Alternatively, the subcommand conda env list displays a list of all environments on your current system; the currently activated one is marked with an asterisk in the middle column. The subcommands of conda env (sometimes with suitable switches) encompass most of your needs for working with environments.

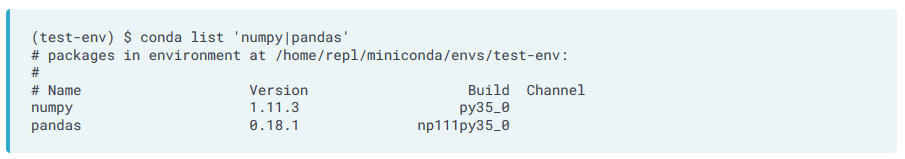
The output of conda env list shows that each environment is associated with a particular directory. This is **not** the same as your current working directory for a given project; being "in" an environment is completely independent of the directory you are working in. Indeed, you often wish to preserve a certain Conda environment and edit resources across multiple project directories (all of which rely on the same environment). The environment directory displayed by conda env list is simply the top-level filepath in which all resources associated with that environment are stored; you need never manipulate those environment directories directly (other than via the conda command); indeed, it is much safer to leave those directories alone!

For example, here is output you might see in a particular terminal:

Following the example above, what is the name of the environment you are using in the current session? Even if you determine the answer without running a command, run conda env list to get a feel of using that subcommand.

## What packages are installed in an environment? (I)

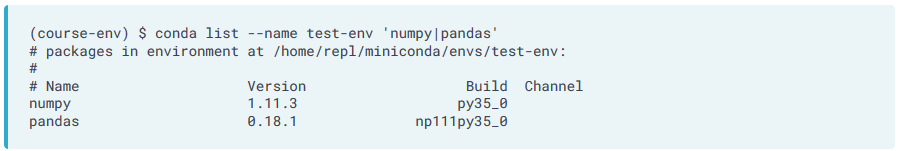
The command conda list seen previously displays all packages installed in the current environment. You can reduce this list by appending the particular package you want as an option. The package can be specified either as a simple name, or as a regular expression pattern. This still displays the version (and channel) associated with the installed package(s). For example:

Without specifying 'numpy|pandas', these same two lines would be printed, simply interspersed with many others for the various other installed packages. Notice that the output displays the filepath associated with the current environment first: in this case, /home/repl/miniconda/envs/test-env as test-env is the active environment (as also shown at the prompt).

Following this example, what versions of numpy and pandas are installed in the current (base/root) environment?

## What packages are installed in an environment? (II)

It is often useful to query a different environment's configuration (i.e., as opposed to the currently active environment). You might do this simply to verify the package versions in that environment that you need for a given project. Or you may wish to find out what versions you or a colleague used in some prior project (developed in that other environment). The switch --name or -n allows you to query another environment. For example,

Without specifying the --name argument, the command conda list would run in the current environment. The output would then be the versions of numpy and pandas present in the current environment.

Suppose you created an environment called pd-2015 in 2015 when you were working on a project. Identify which versions of numpy and pandas were installed in the environment pd-2015.

## Switch between environments

Simply having different environments is not of much use; you need to be able to switch between environments. Most typically this is done at the command line, using the conda command. With some other interfaces (like Anaconda Navigator or Jupyter with nb\_conda installed), other techniques for selecting environment are available. But for this course, you will learn about command-line use.

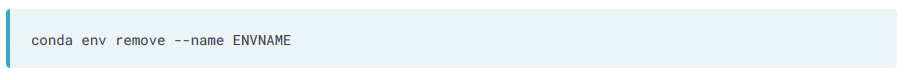
To activate an environment, you simply use **conda activate ENVNAME**. To deactivate an environment, you use **conda deactivate**, which returns you to the root/base environment.

If you used conda outside this course, and prior to version 4.4, you may have seen a more platform specific style. On older versions, Windows users would type activate ENVNAME and deactivate, while Linux and OSX users would type source activate ENVNAME and source deactivate. The unified style across platforms is more friendly. Related to the change to conda activate, version 4.4 and above use a special environment called base that is equivalent to what was called root in older versions. However, in old versions of conda you would not typically see an environment listed on the terminal prompt when you were in the root environment.

## Remove an environment

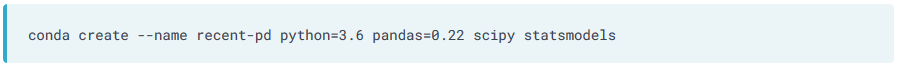
From time to time, it is worth cleaning up the environments you have accumulated just to make management easier. Doing so is not pressing; as they use little space or resources. But it's definitely useful to be able to see a list of only as many environments as are actually useful for you.

The command to remove an environment is:

You may also use the shorter -n switch instead.

## Create a new environment

This course you are working in has come configured with several environments, but in your use you will need to create environments meeting your own purposes. The basic command for creating environments is conda create. You will always need to specify a name for your environment, using --name (or short form -n), and you may optionally specify packages (with optional versions) that you want in that environment intially. You do not need to specify any packages when creating; either way you can add or remove whatever packages you wish from an environment later.

conda create --name recent-pd python=3.6 pandas=0.22 scipy statsmodels

This command will perform consistency resolution on those packages and versions indicated, in the same manner as a conda install will. Notice that even though this command works with environments it is conda create rather than a conda env ... command.

1. Create a new environment called conda-essentials that contains attrs version 15.2 and the best available version of cytoolz (we pick these examples for illustration largely because they are small and have few dependencies).

Conda create –name conda-essentials attrs=15.2 cytoolz

1. Switch into the environment you just created named conda-essentials.

Conda activate conda-essentials

1. Examine all the software packages installed in the current conda-essentials environment.

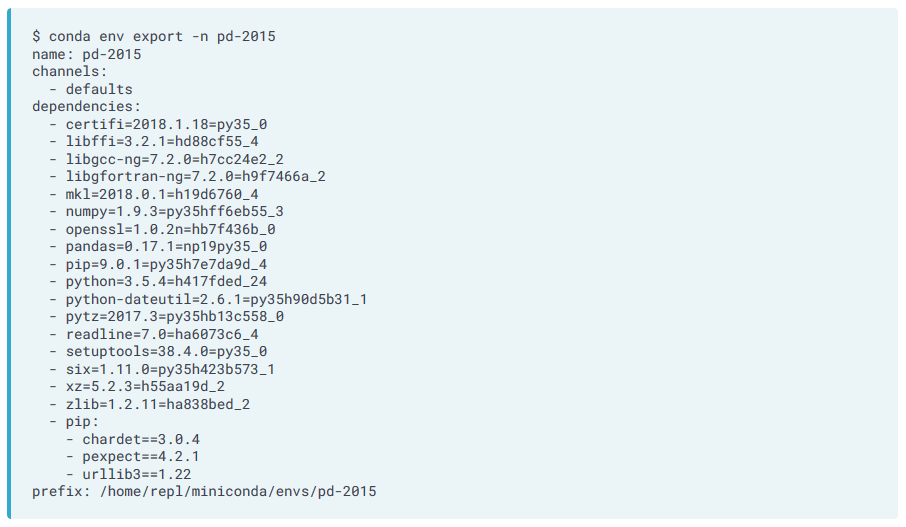
Conda list

## Export an environment

Using conda list provides useful information about the packages that are installed. However, the format it describes packages in is not immediately usable to let a colleague or yourself to recreate exactly the same environment on a different machine. For that you want the conda env export command.

There are several optional switches to this command. If you specify -n or --name you can export an environment other than the active one. Without that switch it chooses the active environment. If you specify -f or --file you can output the environment specification to a file rather than just to the terminal output. If you are familiar with piping, you might prefer to pipe the output to a file rather than use the --file switch. By convention, the name environment.yml is used for environment, but any name can be used (but the extension .yml is strongly encouraged).

Without saving to a file, the output might look similar to the below. Notice that this gives exact versions of packages, not simply ranges or prefixes. This assures **exact** reproducibility of computation within the same environment on a different machine.

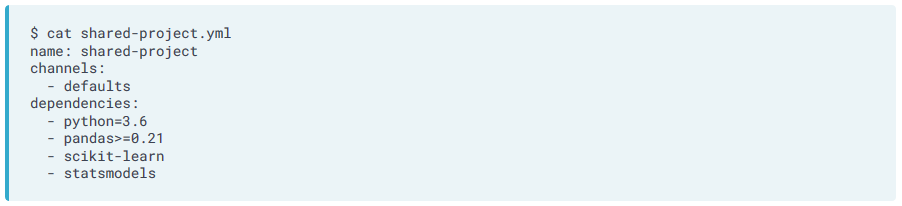


Export the environment called course-env to the file course-env.yml.

Conda env export –n course-env –f course-env.yml

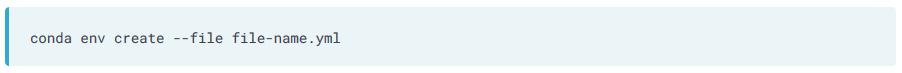
## Create an environment from a shared specification

You may recreate an environment from one of the YAML (Yet Another Markup Language) format files produced by conda env export. However, it is also easy to hand write an environment specification with less detail. For example, you might describe an environment you need and want to share with colleagues as follows:

Clearly this version is much less specific than what conda env export produces. But it indicates the general tools, with some version specification, that will be required to work on a shared project. Actually creating an environment from this sketched out specification will fill in all the dependencies of those large projects whose packages are named, and this will install dozens of packages not explicitly listed. Often you are happy to have other dependencies in the manner conda decides is best.

Of course, a fully fleshed out specification like we saw in the previous exercise are equally usable. Non-relevant details like the path to the environment on the local system are ignored when installing to a different machine—or indeed to a different platform altogether, which will work equally well.

To create an environment from file-name.yml, you can use the following command:

As a special convention, if you use the plain command conda env create without specifying a YAML file, it will assume you mean the file environment.yml that lives in the local directory.

A file environment.yml exists in the local directory within the current session. Use this file to create an environment called shared-project.

conda env create --file environment.yml -n shared-project

# Case Study on Using Environments

## Compatibility with different versions

A common case for using environments is in developing scripts or Jupyter notebooks that rely on particular software versions for their functionality. Over time, the underlying tools might change, making updating the scripts worthwhile. Being able to switch between environments with different versions of the underlying packages installed makes this development process much easier.

In this chapter, you will walk through steps of doing this with a very simple script as an example.

1. The file weekly\_humidity.py is stored in the current session. First just take a look at it using the Unix tool cat. You will see that the purpose of this script is rather trivial: it shows the last few days of the rolling mean of humidity taken from a data file. It would be easy to generalize this with switches to show different periods, different rolling intervals, different data sources, etc.

Cat weekly\_humidity.py

1. Run the script (in the current base environment).

python weekly\_humidity.py

1. The script ran and produced a little report of the rolling mean of humidity. However, it also produced some rather noisy complaints about deprecated syntax in the Pandas library (called a FutureWarning). You now remember that you created this script a few years ago when you were using the pd-2015 environment. Switch to that environment.

Conda activate pd-2015

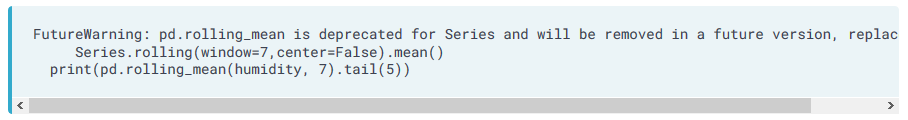
1. Run the script in the current pd-2015 environment. You will notice that the report itself is the same, but the FutureWarning is not present. For a first step, this is how to utilize this script.
2. python weekly\_humidity.py

python weekly\_humidity.py

## Updating a script

You certainly have the easy option to continue using the pd-2015 environment whenever you need to run the weekly\_humidity.py script. Environments can be kept around as long as you like and will assure that all your old scripts (and notebooks, libraries, etc) continue to run the same way they always have.

But quite likely you would like to update your script for your colleagues who use more recent versions of Python. Ideally, you would like them not to have to worry about this message:

Happily, the warning itself pretty much tells you exactly how to update your script