Setting to create and upload conda packages

Step 1: Make an account on https://anaconda.org/

Step 2: Configure your local installation of conda

```
# Optional step
conda config --set anaconda_upload yes # OPTIONAL setting to always upload built
packages to your conda channel

# Install software that handle authenticating interactions with your anaconda
account
mamba install anaconda-client # install in base
anaconda login # provide your login credentials

# Install the software that does the building
mamba install conda-build
```

Step 3: Configure your package build settings

The remainder of this guide will be concerned with writing instructions for `conda-build` so that it correctly creates the package. We will only cover how to configure a fairly simple package. You can find the [documentation for conda-build here] (https://docs.conda.io/projects/conda-build/en/latest/).

As the documentation says:

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

If you want to make a package that differs from what we make in class, the recommended approach is therefore to start somewhere and try things until it works. Once you have made a few packages, you will start to get the hang of how to set up different functionality. However, because of the diversity of different kinds of packages that can be build with conda, there is no one guide to follow to build every package.

For creating any package, the first step is to create the configuration files. You may wish to organize these into a directory and you can git track them in the repo of your package if you wish.

You always need a meta.yaml file (documented here). YAML (yet another markup language) is a structured language for storing data and commands. YAML supports sections with subsections and uses indentation (like that of Python) to indicate which lines are within a section of subsection. YAML is highly flexible and can be used to store a wide variety of data. In the case of conda, there are several sections that can contain settings for different stages or aspects of the build process.

Let's walk through the sections we will use for out magnum opus package.

Variables

First, if you want to define any variables for use in your YAML file, this is done at the top of the document. This isn't necessary, but can make the document more readable if you are including something like a hash or are going to use the same data in multiple places and will want to change those data later (like a software version number). In the case of our package, we might define the following variables at the top of our YAML file. Note that variable assignment looks like `{% set <variable_name> = <data> %}`

```
{% set name = "magnumopus" %}
{% set version = "0.1" %}
```

We will now be able to refer to our variables anywhere in our YAML file. Variables are referred to using the syntax `{{ <variable_name> }}`.

Two notes before we proceed:

1. You don't have to use the name or version number I use above. The name of the package is simply what is used in `mamba install <package>` commands. This doesn't control how your package is used once installed. It's just what mamba and conda will call it during installation, removal, etc. You can name it something else if you like.

2. You can version however you want. Symantic versioning has guidelines but few fixed rules. The only important rule is that newer versions have higher numbers. Otherwise, the typical naming convention is "major.minor.build" where major versions typically have compatability breaks between them, minor versions typically introduce new functionality, and build versions are bugfixes or minor changes with no user-facing impact. You can increment any of these whenever you like, but it can be helpful to take advantage of the hierarchy of those levels to communicate to users when you are making small vs big changes.

   Some software uses versining with basic major, minor, and build increments. Others use something different. For example, Python uses something similar to the system described above, while Nextflow and Ubuntu use "Year.Month.build". Date-based builds are valuable when you are on a fixed release schedule, while the Python system can be better suited to development where changes are made less regularly or at a less fixed pace. It's up to you how you want to version your software.

Package details

Next we define the Major details of our package. These are the name and version of the package build. As we defined variables above, we can use those. Otherwise you can write these out. Up to you.

```
package:
  name: {{ name|lower }}
  version: {{ version }}
```

Note that YAML supports operations like case-conversion. In the above example, it is converting the contents of the "name" variable to lowercase.

Source

Once you've named your package, you need to tell conda where it is. The source sections specifies the location of the files that should be used to construct your package. This is typically the relative path on your computer or the URL and branch of a git repo.

If you are building a conda package from a local source and you have created a directory to store your conda build files, then your source would looks like this

```
source:
    path: ../
```

If, on the other hand, you are building from a git repo, then you would put something like this

```
source:
    git_url: "https://github.com/User-Name/magnum_opus.git"
    git_rev: "v0.1" # or branch name if you aren't just building the current HEAD commit of the master branch; optional
    git_depth: 1 # Only clones current state of repo, not commit history. Use this if your git_rev is the current commit. If your git_rev is not current, you'll get an error.
```

There are other sources from which you can build. Check out the documentation for some others.

Build

For simple packages like ours that have short build processes, you can specify the command to build the package in your meta.yaml. For more complex builds, you can instead write a build.sh script in the same location as your meta.yaml file which conda will automatically run.

Our build section will look something like this

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

In that script, `$PYTHON` is refering to a build variable that conda has set which points to the Python executable being used. There are several build-specific environment variables that can be used during the build process and referred to in the meta.yaml file.

Requirements

Next comes the requirements section. This includes all the actual configuration of what your package needs in order to run. This sections is typically composed of multiple subsections. Each section defines the requirements of a step of the build process: building, installation (called "host"), and running.

The build section and host section are related in that they both refer to how the package is created. The build section can be thought of a place to specify things specific to the computer (i.e., architecture) on which the package is being built. The host section is where to specify things specific to the computer on which the package will be used. In our case, our package is just a Python script so there aren't architecture-specific requirements. If we were making a package in a compiled language, then there may be things we would need to specify about the source and destination architectures.\

Our host sections will look something like this:

```
requirements:
  host:
    - python >=3.10
    - pip
    - setuptools
```

That host section specifies the Python version and the installation tools that need to be on the computer on which our package is installed. This is the beginning of the specification of the things that conda will install alongside our package. The remainder of those is in the "run" section, which specifies all the software needed to actually use our package.

Our run section will look something like this:

```
  run:
    - python >=3.10
    - numpy # If you used it. Specify version number if there is a version-
specific functionality the package needs
    - pandas # if you used it
    - bioconda::blast # This specifies the channel where the package should be
found in case it is more than one channel
    - bioconda::seqtk
```

In the requirements section you can specify the source channel and version requirements of each dependency. For example, you can specify a range like that in the example above, a range between two versions (python >=3.8,❤️.12), or a specific version (blast==2.13.0). If you specify versions, make sure that all specified packages are compatible. i.e., that they do not have conflicting dependencies. If there are no compatible versions for shared dependencies of the packages specified here, then your package will either not build or will be uninstallable.

Step 4: write the installation script for your package

If you have a single Python script that makes up your package, then you can specify that as a file in an outputs sections like this

```
outputs:
    files: my_script.py # relative path from location specified in source section
```

If your package involves a custom module that must be imported (like our magnum opus), then you will need to pip install your module so that it can be imported from any location. Alternatively, you could tell conda to copy your package directory to the bin directory of the conda environment. However, for this example we will pip install.

Installing custom Python packages simply involves writing a setup.py file that Python's built-in installation tools will use to perform the installation steps. The setup.py file is typically simple and looks something like the following

```
from setuptools import setup

setup(
   name='magnumopus',
   version='0.1',
   description='The culmination of a semester of hard work',
   author='Person Name',
   author_email='Pname42@gatech.edu',
   packages=['magnumopus'],  #same as name
   package_dir={"": "./"}, # Where is the package
   python_requires='>=3.10', # if relevant
   #install_requires=['some_package'], #external packages as dependencies if
relevant
   scripts=['amplicon_align.py'], # path to whatever file has the command line
interface designed to be used by a person

)
```

conda convert

```
conda convert --platform osx-64 /path/to/builds/linux-64/package.tar.bz2 -o
/path/to/builds/
anaconda upload /path/to/builds/osx-64/package.tar.bz2
```