Cheat Sheets

Andrés García Escovar

November 15, 2024

Contents

1 "conda" Cheatsheet

1.1 Create Environment

- Create an empty conda environment:

```
conda create --name <env_name>
```

- Create a conda environment with packages:

```
conda create --name <env_name> <package_name_0>...<package_name_N>
```

- Create a conda environment from a yaml file:

```
conda env create --file <path_to_yaml_file>
```

Sample yaml file:

```
name: myenv
channels:
    - conda-forge
    - defaults
dependencies:
    - python=3.7
    - numpy
    - matplotlib
    - pip
    - pip:
        - tensorflow==2.0.0
```

- Create at a specific location:

```
conda create --prefix <path_to_env>
```

The previous options still hold.

1.2 Activate Environment

- Activate an environment in the main installation directory:

```
conda activate <env_name>
```

- Activate an environment in a specific location:

```
conda activate <path_to_env>
```

- To deactivate an environment:

```
conda deactivate
```

- Activate conda environment in a Python script:

```
import subprocess

script = (
    ". /home/hp/miniconda3/etc/profile.d/conda.sh && conda activate "
    "testenv && conda env list"
)

subprocess.run(script.split(" "), shell=True)
```

  Things to remember:

  - The `shell=True` option is important; with `shell=False`, the code will **not** run.

1.3 Clone Environment

- Clone an environment:

```
conda create --name <env_name> --clone <env_to_clone>
```

1.4 Remove Environment

- Remove an environment:

```
conda remove --name <env_name> --all
```

- Remove and environment in a specific location:

```
conda remove --prefix <path_to_env> --all
```

1.5 Add Packages

Notice that different commands can be mixed and matched.

- Add packages to an environment:

```
conda install --name <env_name> <package_name_0>...<package_name_N>
```

- Add packages to an environment in a specific location:

```
conda install --prefix <path_to_env> <package_name_0>...<package_name_N>
```

- Add packages to an active environment:

  1. Activate the environment (click here).
  2. Add packages:

     ```
     conda install <package_name_0>...<package_name_N>
     ```

- Add packages from a different channel in an active environment:

```
conda install --channel <channel_name> <package_name_0>...<package_name_N>
```

- Add packages with a specific version:

```
conda install <package_name>=<version>
```

- Add packages from a yaml file:

```
conda env update --file <path_to_yaml_file>
```

- Update the packages to the given ones:

```
conda env update --file <path_to_yaml_file> --prune
```

Sample yaml file to udpate:

```
channels:
    - conda-forge
    - defaults
dependencies:
    - python=3.7
    - numpy
    - matplotlib
    - pip
    - pip:
        - tensorflow==2.0.0
```

1.6 Remove Packages

- Remove packages from an environment:

```
conda remove --name <env_name> <package_name_0>...<package_name_N>
```

- Remove packages from an environment in a specific location:

```
conda remove --prefix <path_to_env> <package_name_0>...<package_name_N>
```

- Remove packages from an active environment:

1. Activate the environment (click here).
2. Remove packages:

```
conda remove <package_name_0>...<package_name_N>
```

2 “electron” Cheatsheet

2.1 Package an Application

2.1.1 Installation

It will be assumed that the user already has a working `electron` application. These steps must be followed in the given order:

1. Make sure you have `rpm` installed:

```
sudo apt install rpm
```

2. Navigate to the root directory of the application, i.e., where the `main.js` file is located, that will be the place where the `package.json` file will be created:

```
cd /path/to/app
```

3. install the `electron-forge` package to use in the command line:

```
npm install -D @electron-forge/cli
```

4. Initialize the `electron-forge` project:

```
npm exe -package=@electron-forge/cli import -c "electron-forge import"
```

   This will create a `forge.config.js` file and will add entries to the `package.json` file; all needed to package the application.

5. In the `package,json` file, make sure that the `scripts` section includes the commands to start the application packaging:

```
"scripts": {
    "start": "electron-forge start",
    "package": "electron-forge package",
    "make": "electron-forge make"
}
```

6. Make sure that the `@electron-forge/plugins-fuses` package is installed:

```
npm install -D @electron-forge/plugins-fuses
```

   This will install more configuration files that will be needed to package the application.

7. Install the `electron-squirrel-startup` package:

```
npm install electron-squirrel-startup
```

   This must be a required package for all projects that need to be packaged; notice how there is no `--save-dev` or `-D` flag.

8. In the `forge.config.js` file make sure that the name of the authors and description of the application are included:

```
module.exports = {
    ...
    makers: [
        {
            name: "@electron-forge/maker-squirrel",
            config: {
                name: "Jane Doe",
                description: "My Electron Application",
            }
        },
        ...
    ],
    ...
}
```

9. Package the application:

```
npm run make
```

   This will create a `out` directory with the packaged application. If there are any errors, they will be displayed in the terminal. These might include missing dependencies or incorrect configurations that must be fixed before the application can be packaged.

10. In Linux, the application is typically located in the `out/make/deb/x64` directory.

11. To install the application, navigate to the directory where the `.deb` file is located and run:

```
sudo dpkg -i <app-name>.deb
```

    when the installation is complete, the application will be available in the system's applications menu.

3 “git” Cheatsheet

3.1 Cloning

- Clone a repository:

```
git clone <url>
```

  For example:

```
git clone git@github.com:Python-World/example.git
```

- Clone a repository from a specific branch:

```
git clone -b <branch> --single-branch <url>
```

  For example:

```
git clone -b demo --single-branch git@github.com:Python-World/example.git
```

- Clone a repository with submodules:

```
git clone --recurse-submodules <url>
```

  For example:

```
git clone --recurse-submodules git@github.com:Python-World/example.git
```

3.2 Deleting

- Delete a local branch:

```
git branch -d <branch>
```

  For example:

```
git branch -d demo
```

- Delete a remote branch:

```
git push origin --delete <branch>
```

  For example:

```
git push origin --delete demo
```

3.3 Updating

- Update the list of remote branches:

```
git remote update origin --prune
```

- Update branches with conflicts:

  1. Checkout the branch to be updated, call it `<branch_1>`:

     ```
     git checkout <branch_1>
     ```

  2. Update the local branch, i.e., update `<branch_1>`:

     ```
     git fetch origin
     ```

  3. Merge merge `<branch_2>` into `<branch_1>`:

     ```
     git merge origin/<branch_2>  # branch 2 -> branch 1
     ```

  4. Solve conflicts.:

     ```
     git mergetool  # Select the changes to keep.
     ```

  5. Commit the changes:

     ```
     git commit -m "Solved conflicts."
     ```

3.4 Submodules

- Add a submodule:

```
git submodule add <url>
```

  For example:

```
git submodule add git@github.com:Module-Folder/example.git
```

- Add a submodule with a custom name:

```
git submodule add --name <name> <url> <path>
```

  For example, we want to add a module named `example`, in the `tests` directory, that is located in the current directory:

```
git submodule add tests git@github.com:Python-World/tests-example.git ./tests
```

- Update submodules:

```
git submodule update --init --recursive
```

- Remove a submodule:

  1. Delete the relevant section from the `.gitmodules` file.
  2. Delete the relevant section from `.git/config` directory.
  3. Run `git rm --cached <path_to_submodule>` (no trailing slash).

4. Commit and delete the now untracked submodule files.

- Updating a submodule URL:
  - Change the URL in the `.gitmodules` file:

```
[submodule "example/module]
    path = example/module
    url = ../../Libraries/internal/module.git
```

    that is, change the `url` field.
  - Sync the submodule:

```
git submodule sync
```

  - Update the submodule:

```
git submodule update --init --recursive
```

  - Commit the changes:

```
git commit -m "Updated submodule URL."
```

    at this point, the submodule URL has been updated.

3.5 Tracking

- Create a local branch:

```
git checkout <local_branch>
```

  add the `-b` flag to switch to the branch inmediately after creating it:

```
git checkout -b <local_branch>
```

- Branch from target branch to another branch:

```
git checkout <new_branch> <target_branch>
```

  add the `-b` flag to switch to the branch inmediately after creating it:

```
git checkout -b <new_branch> <target_branch>
```

- Track a remote branch:

```
git branch -u origin/<branch>
```

4 "Make" Cheatsheet

4.1 General Structure

The general structure of a "make" file is as follows:

```
target: pre-requisites
    command
    command
    .
    .
    .
    command
```

- target: is the name of the file generated by the "make" file.
- pre-requisites: are the files used to generate the target. These pre-requisites are also called dependencies. Dependencies will be re-compiled if they have been modified since the last time the target was generated.
- command: is the command used to generate the target. There can be multiple commands per target.

4.2 Using a "makefile"

- To run a simple "makefile" use the file structure shown below:

  ```
  <root>
  |___makefile
  |___main.c
  ```

  The content of the "makefile" is as follows:

  ```
  all: main.c
      gcc -o main main.c
  ```

  Use the file to compile the "main.c" file:

  ```
  $ make
  ```

- To include multiple commands add more targets:

  ```
  default: main.c
      gcc -o main main.c

  clean:
      rm main
  ```

  When running a "makefile" with multiple targets, the first target is the default target. To run a specific target use:

  ```
  $ make clean
  ```

- To include multiple pre-requisites add them to the target:

```
default: main.c main.h
    gcc -o main main.c
```

4.3 Using multiple dependencies

- The following "makefile" runs the three targets to build a program:

```
blah: blah.o
    gcc blah.o -o blah

blah.o: blah.c
    gcc -c blah.c -o blah.o

blah.c:
    echo "int main() { return 0; }" > blah.c
```

4.3.1 How it works

1. Make selects the target "blah", because it is the first target in the "makefile", i.e., the default target.
2. "blah" depends on "blah.o", so make looks for the target "blah.o".
3. "blah.o" depends on "blah.c", so make looks for the target "blah.c".
4. "blah.c" is a phony target, so make runs the command associated with it.
5. Make then goes back to "blah.o", which is now available, and runs the command associated with it.
6. Make then goes back to "blah", which is now available, and runs the command associated with it.

- If **all** targets need to be run, use the "all" target:

```
all: one two three

one:
    echo "one"

two:
    echo "two"

three:
    echo "three"

clean:
    echo "clean"
```

4.4 Variables

Variables can only be strings.

- To define a variable use := or =; the difference is that := is evaluated at the time of definition, while = is evaluated at the time of use. For example:

```
x := $(shell date)
tt:
    echo $(x)
    sleep 2
    echo $(x)
```

will yield:

```
$ make tt
sex 22 jan 2021 14:56:08 -03
sex 22 jan 2021 14:56:08 -03
```

whereas:

```
x = $(shell date)
tt:
echo $(x)
sleep 2
echo $(x)
```

will yield:

```
$ make tt
sex 22 jan 2021 14:56:08 -03
sex 22 jan 2021 14:56:10 -03
```

Notice that the latter yields different results because the variable is evaluated at the time of use, not at the time of definition.

- Variables can be referenced using "$()" or "${}". For example:

```
x = 1
default:
    echo ${x}
    echo $(x)
```

- The "$@" character is a special character that represents the target. For example:

```
all: one two

one two:
    echo $@

# Is the same as:
# one:
#     echo one
# two:
#     echo two
```

4.5 Use of Wildcards

- Wildcards can be used to select multiple files that match a pattern.
- There are three types of wildcards:
  1. "*" matches any number of characters, including none.
  2. "%" matches any single character, including none.
     - This character can be used in "matching mode", i.e., matches one or more characters in a string. This is called the "stem ".
     - This character can be used in "replacing mode", i.e., replaces the stem with the "replacement"string.
     - Often used in rule definitions and some specific functions.

4.6 Implict Rules

There are some implicit rules when using "makefiles ":

- Compiling a C program: `n.o` is made automatically from `n.c` with a command of the form:

```
$(CC) -c $(CPPFLAGS) $(CFLAGS) $^ -o $@
```

- Compiling a C++ program: `n.o` is made automatically from `n.cc` or `n.ccp` with a command of the form:

```
$(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $^ -o $@
```

- Linking a single object file: `n` is made automatically from `n.o` by running the command:

```
$(CC) $(LDFLAGS) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

  where:

  - `CC`: Program for compiling C programs; default is `cc`.
  - `CXX`: Program for compiling C++ programs; default is `g++`.
  - `CFLAGS`: Extra flags to give to the C compiler.
  - `CXXFLAGS`: Extra flags to give to the C++ compiler.
  - `CXXFLAGS`: Extra flags to give to the C++ preprocessor.
  - `LDFLAGS`: Extra flags to give to compilers when they are supposed to invoke the linker. For example, `-L`.

4.7 Important Comments

- When indenting, use tabs, not spaces. Indenting in "make" is done with tabs, not spaces, i.e., tabs are mandatory.

5 "REACT" Cheatsheet

5.1 Installation

To install REACT in Linux we can use the command line:

1. Install the `npm` package manager:

```
sudo apt install npm
```

2. Install the `create-react-app` package:

```
sudo npm install -g create-react-app
```

3. It is recommended to install the `node.js` package:

```
sudo apt install nodejs
```

4. Check the installation:

```
create-react-app --version
```

5.2 Creating an Application

To create and run an application do:

1. Navigate to the directory where you want to create your application:

```
cd <path_to_directory>
```

2. Create the application by using the `create-react-app` command:

```
npx create-react-app <app_name>
```

   the `npx` command a package runner tool that comes with `npm` 5.2+ and higher.

5.3 Running an Application

Once the application has been created:

1. Navigate to the application directory:

```
cd <app_name>
```

2. Run the application:

```
npm start
```

3. Open a web browser and navigate to:

```
http://localhost:3000/
```

6 "Snippets" Cheatsheet

6.1 Conda - .bashrc conda Initialization

The following code initializes the "conda" environment in the ".bashrc" file. It also deactivates the "conda" environment.

```
# >>> conda initialize >>>
_conda_setup="$('<path_to_environment>/bin/conda' 'shell.bash' 'hook' 2> /dev/null)"
if [ $? -eq 0 ]; then
    eval "$_conda_setup"
else
    if [ -f "<path_to_environment>/etc/profile.d/conda.sh" ]; then
        . "<path_to_environment>/etc/profile.d/conda.sh"
    else
        export PATH="<path_to_environment>/bin:$PATH"
    fi
fi
unset _conda_setup
# <<< conda initialize <<<

# Deactivate conda.
conda deactivate
```

example:

```
# >>> conda initialize >>>
_conda_setup="$('/home/hp/miniconda3/bin/conda' 'shell.bash' 'hook' 2> /dev/null)"
if [ $? -eq 0 ]; then
    eval "$_conda_setup"
else
    if [ -f "/home/hp/miniconda3/etc/profile.d/conda.sh" ]; then
        . "/home/hp/miniconda3/etc/profile.d/conda.sh"
    else
        export PATH="/home/hp/miniconda3/bin:$PATH"
    fi
fi
unset _conda_setup
# <<< conda initialize <<<

# Deactivate conda.
conda deactivate
```

6.2 Linux - Get Files from HTTP/FTP Site

To get files from an FTP or HTTP site, a request can be made using the `wget` command:

```
wget --recursive --no-parent <url>/<valid_file/directory_name>
```

6.3 Linux - Untar tar.gz Files

To untar files with the `tar.gz` extension, the following command can be used:

```
tar -xvzf <file.tar.gz>
```

The flags mean:

- `-x`: Extract files from an archive.
- `-v`: Verbose mode, print the names of the files as they are extracted.
- `-z`: Filter the archive through gzip.
- `-f`: Use archive file or device ARCHIVE.

6.4 Python - Live Elapsed Time

For a subprocess:

```
"""
    Code to run a live timer for a subprocess.
"""

# ------------------------------------------------------------------------------------------
# Imports
# ------------------------------------------------------------------------------------------

# General
import subprocess

from datetime import datetime

# ------------------------------------------------------------------------------------------
# Variables
# ------------------------------------------------------------------------------------------

# Time variables.
time_ela = None  # Elapsed time.
time_str = None  # Time at which the process starts.

# Other
command = ["sh", "script.sh"]  # Command to run, this is an example.
elapsed = None  # Elapsed time counter.
flag_process = True  # Flag to indicate if the process is running.
process = None  # Contains the thread of the process.

# ------------------------------------------------------------------------------------------
# Algorithm
# ------------------------------------------------------------------------------------------

# Initial time is always 00:00
print(f"\rElapsed time: {int(0):02d}:{int(0):02d}", end="")

# Start infinite loop.
while True:
    # Only if the process is running.
    if not flag:
        elapsed = (datetime.now() - time_str).total_seconds()
        minutes = elapsed / 60
        seconds = (minutes - int(minutes)) * 60
        print(f"\rElapsed time: {int(minutes):02d}:{int(seconds):02d}", end="")
```

```
    # Only spawn one process.
    if flag:
        # Set the flag to False, start timer and process.
        flag = False
        str_time = datetime.now()
        proc = subprocess.Popen(
            command, stdout=subprocess.PIPE, stderr=subprocess.STDOUT
        )
        continue

    # Exit the loop if the process is done.
    if proc.returncode is not None:
        fstring = f"\rElapsed time: {int(minutes):02d}:{int(seconds):02d}"
        break

# Print the final time.
print(f"\r{fstring}")
```

Things to Consider

- Remember to include a carriage return so the string is always removed.
- Include the carriage return at the ***start*** of the string, ***not*** at the end.

7 "Installations and Setup" Cheatsheet

7.1 Linux - NeoVIM

7.1.1 Installation

1. To install **NeoVIM** there are many methods. The suggested method is to use the *snap* store, or equivalent, by using the command:

   ```
   $ sudo snap install nvim --classic
   ```

   This will install **NeoVIM** in the system.

2. The *–classic* flag is used to install the software with full access to the system, which is required for some plugins and, most of the time, for the software to even install.

3. Make sure that the *.bashrc* has the snap store path in the `PATH` variable at the start of the list:

   ```
   export PATH=/snap/bin:$PATH
   ```

4. To make sure the installation was successful, run the command:

   ```
   $ nvim --version
   ```

   Make sure that this is not a "developer version"; i.e., the latest stable version.

7.1.2 Setup - Lazy: Package Manager

This procedure is to perform the basic setup for Lazy and get **NeoVIM** running in a basic way.

1. Make sure that **NeoVIM** is properly installed.

2. If the directory `~/.config/nvim` does not exist, create it:

   ```
   $ mkdir -p ~/.config/nvim
   ```

3. Create the file `~/.config/nvim/init.lua` file in the directory:

   ```
   touch ~/.config/nvim/init.lua
   ```

   This is done for the modern version of **NeoVIM**, which uses the *lua* language for configuration.

4. Create the `~/.config/nvim/lua` directory, if it does not exist:

   ```
   mkdir ~/.config/nvim/lua
   ```

5. In the `~/.config/nvim/lua` directory, create the three directories:

   - config
   - plugins
   - util

   This is where the different configuration files will be stored. Since this is the default place where **NeoVim** will look for the different files and directories, it is a good idea to keep the default structure.

6. At this point, the file and directory structure should look like:

```
<root == ~/.config/nvim>
| init.lua
|_ lua
    |_ config
    |_ plugins
    |_ util
```

7. To dowload the packages and give them the basic configuration, create an `init.lua` in the `config` directory:

```
<root == ~/.config/nvim>
| init.lua
|_ lua
    |_ config
    |   |_ init.lua
    |_ plugins
    |_ util
```

8. Open the file `<root>/lua/config/init.lua` file and add the following lines:

```lua
local lazypath = vim.fn.stdpath("data") .. "/lazy/lazy.nvim"
if not vim.loop.fs_stat(lazypath) then
  vim.fn.system({
    "git",
    "clone",
    "--filter=blob:none",
    "https://github.com/folke/lazy.nvim.git",
    "--branch=stable", -- latest stable release
    lazypath,
  })
end
vim.opt.rtp:prepend(lazypath)

-- Example using a list of specs with the default options
-- Make sure to set `mapleader` before lazy so your mappings are correct
vim.g.mapleader = " "

require("lazy").setup({
  "folke/which-key.nvim",
  { "folke/neoconf.nvim", cmd = "Neoconf" },
  "folke/neodev.nvim",
})
```

9. Use **NeoVIM** to open the file and, in **NeoVIM** command console mode (press esc), run the command: command:

```
~
-------------------------------------
:checkhealth
```

7.2 Miscellaneous - Setup SSH Keys to Connect Between Machines

The instructions are for setting up SSH keys to connect between machines, it should be useful for any Linux distribution, MacOS, and Windows.

For this, we will need **OpenSSH** installed in both the origin and host machines. You can look for the installation instructions for your specific system.

7.2.1 Installation

We want to connect from *machine_0* to *machine_1* without having to type the password every time.

Setup for machine_0:

- Create and ssh key; this key will have a private and public part. The public part will be copied to the receiving machine, i.e., *machine_1*. T he private key will be kept in the sending machine, i.e., *machine_0*:

  ```
  ssh-keygen -t rsa -b 4096
  ```

  This will create a key with 4096 bits of encryption (suggested).

  - To add a comment at the end of the key, use the `-C` flag:

    ```
    ssh-keygen -t rsa -b 4096 -C "Comment here"
    ```

- The command will ask for a location to save the key, make sure to choose the default location, which is `~/.ssh` directory; the name you can choose:

  ```
  Enter file in which to save the key (/home/user/.ssh/id_rsa): /home/user/.ssh/name
  ```

- If you want to add a passphrase to the key, you can do so. This will add an extra layer of security to the key. If you do not want to add a passphrase, just press enter :

  ```
  Enter passphrase (empty for no passphrase):
  ```

- The key will be created and saved in the location you specified. There will be two files: `name` and `name.pub`, where `name` is the private key and `name.pub` is the public key.
- Copy the public key such that it can be added to the receiving machine.
- To make the connection automatic, you can use the file `~/.ssh/config` to setup the connection. This file should look like:

  ```
  Host machine_1
      HostName machine_1
      User user
      IdentityFile ~/.ssh/name
  ```

Setup for machine_1:

- With the public key in hand, copy the contents of the file into the `~/.ssh/authorized_keys` file in the receiving machine:

```
cat name.pub >> ~/.ssh/authorized_keys
```

  - For Windows, in Powershell (admin mode), use the following:

```
Get-Content name.pub >> ~/.ssh/authorized_keys
```

- Make sure that the permissions for the `~/.ssh` directory are set to `700` and the permissions for the `~/.ssh/authorized_keys` file are set to `600`:

```
chmod 700 ~/.ssh
chmod 600 ~/.ssh/authorized_keys
```

  - For Windows, there is no need to change the permissions.