useful_commands

Jaime Abraham Castro-Mondragon

Last update: 2025-06-04

Introduction

In this document you will find many commands from multiple languages, for instance `R`, `python`, `bash`, `Snakemake`, or others. I use these commands frequently and I think they are useful **but easy to forget**.

So, to avoid searching on the `bash history` or in google, it is better to have a repository with all of them, enjoy.

`aws`

`log in`

Run the command and open the browser to confirm

```
aws configure sso
aws sso login
```

`git`

Create a new branch from a previous `commit`

Use `git log` to get the commit hash, `git branch` to create a new branch, and `git switch` to modify the new branch.

Don't forget to double check the branch you are working with using `git status`

```
git log
git branch new_branch_name commit_hash
git switch new_branch_name
```

Merge a `branch` into `main`

Don't forget to double check the branch you are working with using `git status` and list the branches with `git branch`

```
# Go to the branch
git checkout branch_to_merge

# FETCH_HEAD
git pull origin main

# go to main branch and merge
git checkout main
```

```
git merge branch_to_merge

# If needed add the files with differences
git add [resolved_files]

# No -m
git commit

# Push merged branch to main
git push origin main
```

`git tag`

First `git commit` your changes before adding a new `tag` .

```
git commit -m 'Your message' folder/your_updated_file.txt
git push
git tag v1.2.4
git push --tags
```

Set credentials in `git`

Do this to stop `git` asking your user name each time you `push` changes in a repository.

```
# Set your user name
git config --global credential.https://github.com.username user_name

# Then save your credentials
git config --global credential.helper store
```

Examples of `git commit`

These examples follow the git commit style guidelines

Summary:

- Must be present tense
- Written in the imperative
- First letter is not capitalized
- Does not end with a '.'

Allowed Types:

- feat -> feature
- fix -> bug fix
- docs -> documentation
- style -> formatting, lint stuff
- refactor -> code restructure without changing exterrnal behavior
- test -> adding missing tests
- chore -> maintenance
- init -> initial commit
- rearrange -> files moved, added, deleted etc
- update -> update code (versions, library compatibility)

```
# First commit (init)
git commit -m 'init: First commit' your_file.txt
```

```
# Bug fix (fix)
git commit -m 'fix: Fixed bug when reading data in function()' your_file.txt

# Documentation (docs)
git commit -m 'docs: Added documentation for function()' your_file.txt
git commit -m 'docs: Added comments regarding function()' your_file.txt

# Features (feat)
git commit -m 'feat: Added new method to do ... in function()' your_file.txt

# Style (style)
git commit -m 'style: corrected typo in a comment' your_file.txt
git commit -m 'style: changed colors in heatmap' your_file.txt
git commit -m 'style: linten code/function' your_file.txt
git commit -m 'style: updated style in section ...' your_file.txt

# Refactor (refactor)
git commit -m 'refactor: function() refactored to be faster/memory efficient' your_file.txt

# Test (test)
git commit -m 'test: Added unit test for function()' your_file.txt

# Rearrange (rearrange) : all related to file manipulation
git commit -m 'rearrange: added .gitignore'
git commit -m 'rearrange: moved to subfolder_x' your_file.txt
git commit -m 'rearrange: removed file' your_file.txt
git commit -m 'rearrange: changed chmod' your_file.txt

# Update (update)
git commit -m 'update: new version' your_file.txt
git commit -m 'update: updated function() to use dplyr 3.0' your_file.txt

# Maintenance (chore)
git commit -m 'chore: updated config file' your_file.txt
git commit -m 'chore: removed non required dependencies' your_file.txt
git commit -m 'chore: added dplyr as dependency' your_file.txt
```

python

module version

We can use `pip` from the command line .

```
pip show module_name
```

poetry basics

Add, update or remove modules .

```
poetry update module

# Add a module or a specific version or the latest version
poetry add python
```

```
poetry add python=3.8
poetry add python@latest

# Remove a module
poetry remove python
```

Vectorize a function using `numpy`

Use it to vectorize a function and avoid explicit iterations.

```
import numpy as np

# This function counts occurrences of a specific letter in a string
def count_letter(input_string, letter, case_sensitive=True):
    """
    Counts the number of a specific letter in the input string, with an option for case sensitivity.

    Parameters:
        input_string (str): The string to analyze.
        letter (str): The letter to count.
        case_sensitive (bool): If True, respects case. If False, ignores case.

    Returns:
        int: The count of the specified letter in the string.
    """
    if not case_sensitive:
        input_string = input_string.lower()
        letter = letter.lower()
    return input_string.count(letter)

# Vectorize the function
vectorized_count_letter = np.vectorize(count_letter)

# Example usage
example_strings = np.array(["Abracadabra", "Alpha", "array", "Artistic", "awesome"])

# Call the vectorized function and count the occurrences
As = vectorized_count_letter(example_strings,'A', case_sensitive = False)
Bs = vectorized_count_letter(example_strings,'B', case_sensitive = False)

print(f"Counts of As: {As}")
```

```
## Counts of As: [5 2 2 1 1]
```

```
print(f"Counts of Bs: {Bs}")
```

```
## Counts of Bs: [2 0 0 0 0]
```

R

Generate a color palette with `N` colors using `rcartocolor`.

See all the available palettes here.

```r
library(rcartocolor)

#' @description
#' Create a color palette from rcartocolor with N colors.
#' Users can specify the number of seeds colors (default = 10)
#'
#' @param N An integer
#' @param rcc.palette A string.
#' @param seed.colors An integer
#' @return A vector with N Hex color codes
#' @examples
#' generate.color.palette(10)
generate.color.palette <- function(N           = 10,
                                   rcc.palette = "SunsetDark",
                                   seed.colors = 10) {

  ## Cluster colors
  nb.seed.colors <- ifelse(N < seed.colors,
                           yes = N,
                           no  = seed.colors)

  ## Generate a carto palette and expand it to the number of clusters
  carto.pal.classes <- carto_pal(nb.seed.colors, rcc.palette)
  class.colors      <- colorRampPalette(carto.pal.classes, space = "Lab")(N)

  return(class.colors)
}

# Examples
generate.color.palette(5)
```

```
## [1] "#FCDE9B" "#F48570" "#E34E6E" "#D72C7B" "#7B1D6E"
```

```r
generate.color.palette(10)
```

```
##  [1] "#FCDE9B" "#F9B781" "#F69372" "#F0736D" "#E65A6D" "#DF4671" "#DC3977"
##  [8] "#C42B79" "#A42176" "#7B1D6E"
```

Generate a `renv.lock` file using `renv`

This file is needed in `github actions` to manages dependencies and cache `R` packages.

```r
library(renv)
library(rcartocolor)

# initialize a new project (with an empty R library)
renv::init(bare = TRUE)

# install digest 0.6.19
renv::install("digest@0.6.19")

# save library state to lockfile
renv::snapshot()

# remove digest from library
```

```r
renv::remove("digest")

# check library status
renv::status()

# restore lockfile, thereby reinstalling digest 0.6.19
renv::restore()


#  Control dependencies : https://rstudio.github.io/renv/reference/snapshot.html
renv::settings$snapshot.type("explicit")
```

Convert PDF to high resolution images using `convert`

Use this when you have a PDF file and want to get a picture from it, avoiding to take a screenshot with low resolution.

```
# Install imagemagick first
sudo apt-get install imagemagick

convert -density 300 -trim Input_PDF.pdf -quality 100 Output_Picture.jpg
```