

# L1 - Reproducible & Trustworthy Workflow

November 8, 2023 9:44 AM

Syllabus [https://pages.github.ubc.ca/MDS-2023-24/DSCI\\_522\\_dsci-workflows\\_students/](https://pages.github.ubc.ca/MDS-2023-24/DSCI_522_dsci-workflows_students/)

Lecture notes: <https://ubc-dsci.github.io/reproducible-and-trustworthy-workflows-for-data-science/README.html>

Group project: 1 milestone/week

Watch video on Canvas before class

---

**Data science:** the study, development and practice of reproducible and auditable processes to obtain insight from data.

**Reproducible analysis:** reaching the same result given the same input, computational methods and conditions.

- Input = data
- Computational methods = computer code
- conditions = computational environment (programming language & it's **dependencies**)

**Dependencies** are other things one need to install to run your code, and includes:

- programming languages (e.g., R, Python, Julia, etc)
- packages from programming languages (e.g., tidyverse, scikit-learn)
- other tools you rely on (e.g., Make)
- legacy code (e.g., perl scripts, fortran, etc)
- Dependencies include versions as well as names!

It is important that insights from data science are **trustworthy**: it should be reproducible, auditable, correct, complete, fair, equitable and honest.

- **Auditable/transparent analysis:** a readable record of the steps used to carry out the analysis as well as a record of how the analysis methods evolved

**Complex project** has at least one of the following:

- two, or more, people directly working on the analysis
- involve two or more coding documents
- involve analysis of medium/large data
- Projects where you are working on a remote machine
- have many software or environment dependencies, or ones that are difficult or take a long time to install.
- As a project accumulates more of these features it grows further in complexity.

Workflow features to mitigate chaos:

1. **Version control.** Git & Github

*\*always add version control to workflow!*

2. **Executable analysis scripts & pipelines** (Python/R scripts & Make): Can be used by others to run/replicate the analysis. Easier to understand the landscape of the project and for others to contribute. Reduces some of the challenges/frustrations of working with larger data sets.

*\*add this when you start hiding code chunks/cells in RMD/Jupyter notebook.*

**Make:** a tool that automatically builds executable programs and libraries from source code by reading files called Makefiles. Make can track changes in the source code and its dependencies, only rebuilding the parts of the software that have been modified.

**Makefiles** are "smart" and after changes, only run the parts of the analysis that has changed. Each code block = a rule. Makefiles are made of many rules, typically one rule for each time you run an analysis script.

3. **Defined & shippable dependencies** (Docker)

*\*add this when remote computing or when have tricky dependencies*

**Defined Dependencies** = specific components a software application relies on to function correctly.

**Shippable Dependencies**= the dependencies that are included with the software package when it is ready for deployment or "shipping" to end-users.

**Docker:** a platform that enables developers to build, package, and distribute applications as lightweight, portable containers. **Containers** are isolated environments that contain everything needed to run an application, including the code, runtime, system tools, system libraries, and settings.

**Docker Hub:** a cloud-based repository service provided by Docker that allows users to store and manage Docker images. **Docker images** = key building blocks used to create Docker containers.

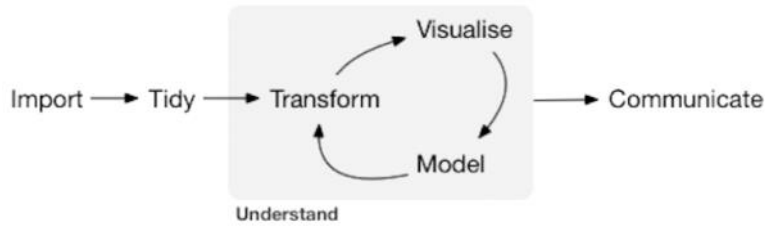
**Instructions needed to run analysis on almost any machine:**

- o Install Docker
- o Clone or download [the GitHub repository](#)
- o From the root of the cloned repository, type:

`docker run --rm -v $(pwd):/home/rstudio/data_analysis_eg \`

```
tttimbers/data_analysis_pipeline_eg make -C /home/rstudio/data_analysis_eg all
```

## Data Analysis Cycle



To drive insight from data, need to match the correct DS methods to type of statistical questions you are asking:

### Statistical questions

- **Descriptive:** summarize a characteristic of a set of data.

Ex: What is the frequency of bacterial illnesses in a set of data collected from a group of individuals? How many people live in each US state?

- **Exploratory:** analyze a set of data to find patterns, trends or relationships between variables.

Ex: Do diets rich in certain foods have differing frequencies of bacterial illnesses in a set of data collected from a group of individuals?

- **Inferential:** analyze the data to see if there are patterns, trends, or relationships between variables in a representative sample.

Ex: Is eating at least five servings a day of fresh fruit and vegetables associated with fewer bacterial illnesses per year?

- **Predictive:** predict measurements or labels for individuals. Less interested in causes.

Ex: How many viral illnesses will someone have next year?

- **Causal:** whether changing one factor will change another factor.

Ex: Does smoking lead to cancer?

- **Mechanistic:** explain the underlying mechanism of the observed patterns, trends, or relationship.

Ex: How do changes in diet lead to a reduction in the number of bacterial illnesses?

### Deal with merge conflict:

**merge conflict** happen when you work collaboratively with teammates on one git repo.

- **Push conflict:** Always pull before push. If you push when there are new changes not been pulled, you will have a message in terminal like this:

```
To https://github.com/joelostblom/DSCI_521_lab1_jostblom
! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to
'https://github.com/joelostblom/DSCI_521_lab1_jostblom'
hint: Updates were rejected because the tip of your current branch
is behind
hint: its remote counterpart. Merge the remote changes (e.g. 'git
pull')
hint: before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for
details.
```

- **Pull conflict (merge conflict):** 2 scenarios when try to pull:

- ①. Changes to a document where **different lines are modified** between remote repo and your local repo. Git can merge when you pull.
- ②. Changes to a document where **same lines are modified** between remote repo and your local repo. **Git CANNOT merge these and will raise a "merge conflict" when you pull.**

```
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 3 (delta 1)
Unpacking objects: 100% (3/3), done.
From https://github.com/joelostblom/DSCI_521_lab1_jostblom
* branch          master      -> FETCH_HEAD
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

Above the conflict happens at README.md, can use **code README.md** to open it in VS code to edit.

- **Approaches to merge conflict:**

- ①. **Edit the conflict file yourself after pull:** Git will tell you where the conflict is in the output of git pull. You then need to edit the conflict content of the file (search for <<<<< HEAD) to make sure it looks good. Then add and commit the updated version.

Ex: conflict content of a text file as below. Edit the two conflict lines, remove any extra characters, **then add and commit and push again.**

```

Some text up here
that is not part of the merge conflict,
but just included for context.
<<<<<< HEAD
We added this line in our last commit
=====
This line was added somewhere else
>>>>>> dabb4c8c450e8475aee9b14b4383acc99f42af1d

```

- <<<<<< HEAD precedes the change you made (that you couldn't push)
- ===== is a separator between the conflicting changes
- >>>>>> dabb4c8c450e8475aee9b14b4383acc99f42af1d flags the end of the conflicting change you pulled from GitHub

- ②. **Stashing local non-committed changes before pulling:** if made changes to a file locally but haven't push yet, and that same file has also been changed remotely. To synchronize changes, using **git pull** directly will result in a **conflict**. So, in this case, using **git stash** to temporarily save all your local changes(both committed + uncommitted), then use **git pull** to update local repo, and later reapply your changes using **git stash pop** to put your changes back to the working area, and then carry on working.

## Dealing with branches

In terminal under local repo, create new branch:

- # Create a new branch : **git branch branch\_name**
- # Switch to the existing branch: **git checkout branch\_name**
- # Create and Switch to a New Branch: **git checkout -b new\_branch\_name**
- # Switch branch: **git switch branch\_name**

After done editing on local new branch, add, commit change and push to remote repo:

# Create a corresponding remote branch and push the changes to remote repo on that branch.

**git push --set-upstream origin new\_branch\_name**

\*origin = the label/nickname for the remote repo where the local repo is cloned from. Can contain many branches.

After this you will have two branches on remote repo(main, new\_branch), with the new\_branch containing your pushed edition.

To merge changes from new\_branch to main branch:

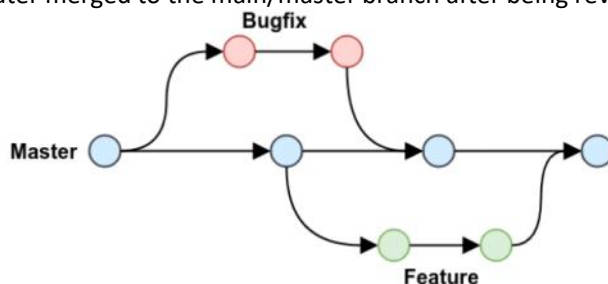
submit a **pull request** on GitHub under the new \_branch. Team members will discuss and double check, after that, the pull request is merged into the main branch.

**Solve conflict between branches** - If there are conflicting changes in the same file between the branch you're trying to merge and the target branch:

1. GitHub will detect and notify by a message if there are conflicting changes between branches after send pull request.
2. To resolve the conflict, need to **manually edit the conflicting file(s) locally**: Open the file(s) in a text editor, and you'll see markers (<<<<<<, =====, and >>>>>>) indicating the conflicting sections. Edit the file to resolve the conflicts
3. After resolving the conflicts, **add, commit, and push** the branch with the conflict resolution back to the remote repository.
4. GitHub will **auto update the pull request status** once it detects the conflict resolution commit.
5. Reviewers can then continue reviewing the pull request, and if everything looks good, the pull request can be merged.

## Git and GitHub flow

**GitHub flow:** a dot= a commit. Every new contribution is done on a new branch (ex: Bugfix, Feature) created from main, and later merged to the main/master branch after being reviewed and tested.

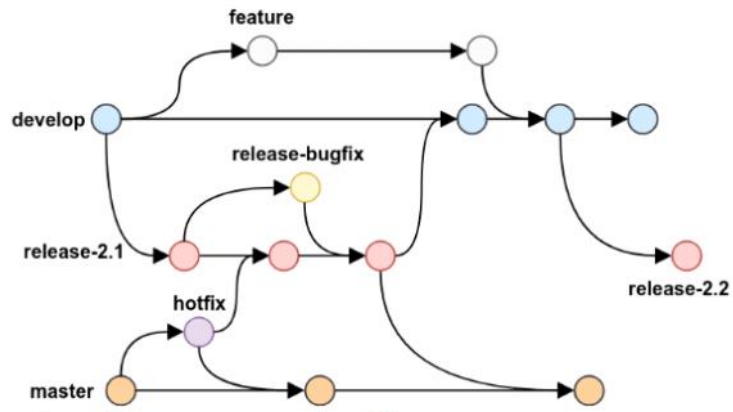


**Git flow:** not recommended for MDS, complicated and time-consuming

- **Two "main" branches:**
  - o Main/master where user is exposed to
  - o Develop/staging where things are tested before they are released to main.
- **Three supporting branches:**
  - o Feature: merged into develop before they are incorporated into a release

- Release: eventually get merged into main after adequate review and testing
- Hotfix.

\*Both feature & release branches are created from develop.



# L2 - Manage dependencies using containerization

November 16, 2023 10:57 PM

<https://ubc-dsci.github.io/reproducible-and-trustworthy-workflows-for-data-science/materials/lectures/05-containerization.html>

**Dependencies** are other things one need to install to run your code, and includes:

- programming languages (e.g., R, Python, Julia, etc)
- packages from programming languages (e.g., tidyverse, scikit-learn)
- other tools you rely on (e.g., Make)
- legacy code (e.g., perl scripts, fortran, etc)

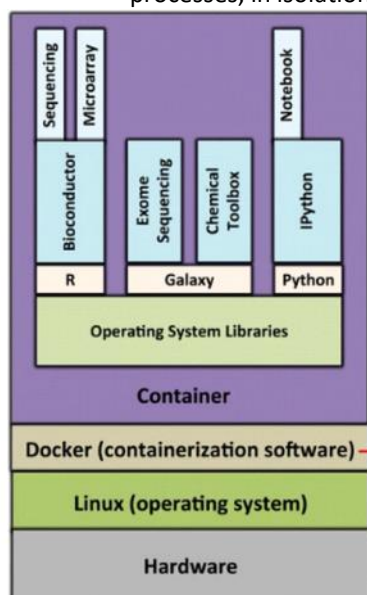
\*Dependencies include versions as well as names

**Defined Dependencies** = specific components a software application relies on to function correctly.

**Shippable Dependencies**= the dependencies that are included with the software package when it is ready for deployment or "shipping" to end-users.

**Containerization:** a technique that encapsulate an application and its dependencies into a standardized unit called a **container**.

- **Virtualization** is a process to divide the elements of a single computer into several virtual elements (computer hardware platforms, storage devices, and computer network resources, and even operating system user spaces)
- **Containers** are isolated environments that contain everything needed to run an application, including the code, runtime, system tools, system libraries, and settings.
  - o Containers virtualize operating system user spaces so that they can isolate the processes they contain, as well as control the processes' access to computer resources (e.g., CPUs, memory and disk space).
  - o An operating system user space can be carved up into multiple containers running the same, or different processes, in isolation.



*does virtualization of operating system user space. allow us to run multiple containers*

\*note that containerization software still use the same operating system and hardware of the computer.

**Docker:** a platform/containerization tool that enables developers to build, package, and distribute applications as lightweight, portable containers.

**Docker Image:** a lightweight, standalone, and executable package that includes all the necessary components to run a piece of software.

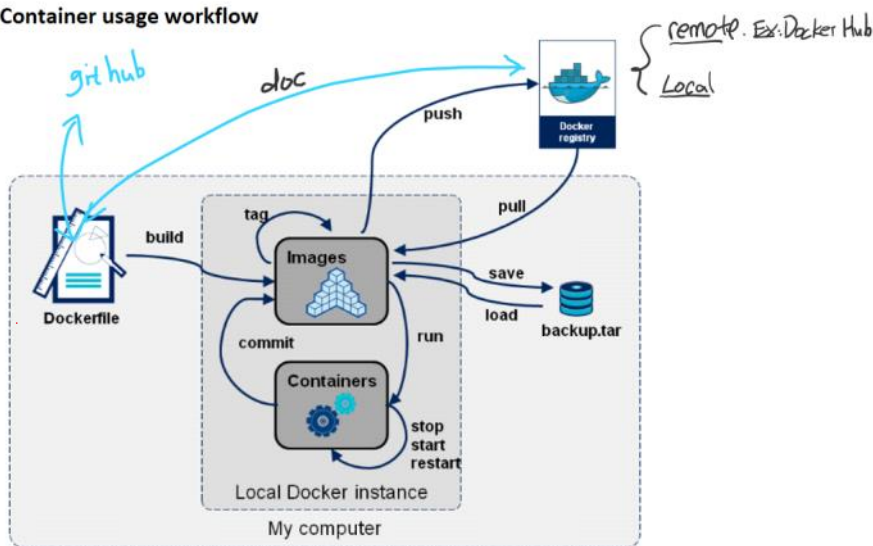
- a snapshot of a file system, containing the application code, runtime, libraries, and system tools.
- immutable and versioned

**Docker Container:** a runnable instance of a Docker image. Can have multiple containers running from the same image, and each container is isolated from others. Ex: image = google chrome app, container=google webpage

An image is a static, immutable package, while a container is a dynamic, runnable instance of that package

**Docker Registry:** a service that stores and distributes Docker images. It can be a **local registry** (hosted on the same machine as the Docker daemon) or a **remote registry** (Ex: **Docker Hub**) accessible over a network.

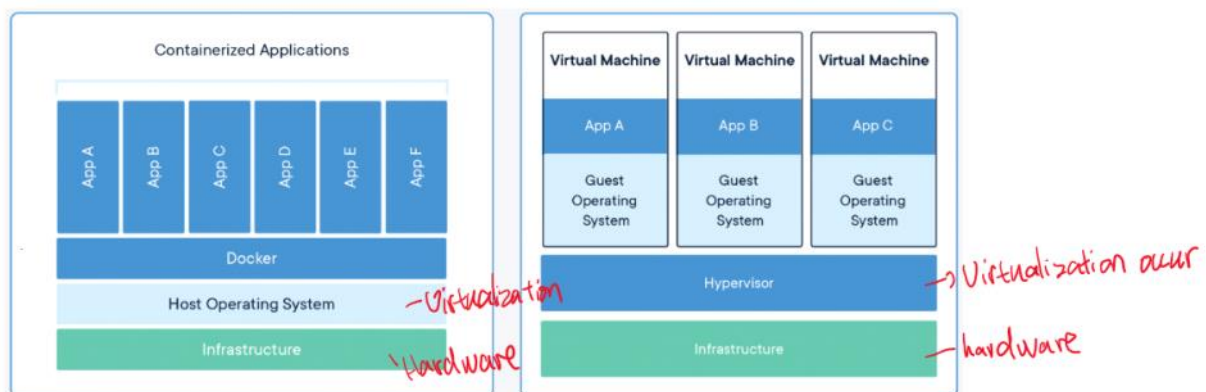
### Container usage workflow



### Contrasting Virtualization Tools: container vs virtual machine

**Virtual machine:** another technology used to generate (and share) isolated computational environments.

- emulate the functionality of an entire computer on another computer. Virtualization occurs at the layer of software that sits between the computer's hardware and the operating system(s). This software is called a **hypervisor**.
- Ex: on a Mac laptop, you can install a program called Oracle Virtual Box to run a virtual machine whose operating system was Windows 10.



Containerization software shares the host's operating system, whereas virtual machines have a completely separate, additional operating system. This can make containers **lighter** (smaller in terms of size) and **more resource and time-efficient** than using a virtual machine.

Virtual environment		Container	Virtual machine
Feature	Application	Operating system user-space	Hardware
Virtualization level	Application	Operating system user-space	Hardware
Isolation	Programming languages, packages	Programming languages, packages, <b>other software, operating system dependencies, filesystems, networks</b>	Programming languages, packages, other software, operating system dependencies, filesystems, networks, <b>operating systems</b>
Size	Extremely light-weight	light-weight	heavy-weight

**Operating system:** the core software that manages computer hardware and provides various services and interfaces for applications and users. Ex: Windows, macOS, Linux, and Unix.

- Bash and the terminal are components that exist within an operating system to facilitate command-line interactions but are not operating systems themselves.
- the system's memory is divided into two main areas: the **user space** and the **kernel space**.

- **User Space:** the area of memory where user applications, programs, and processes operate. User space applications are programs like word processors, web browsers, media players, and other software that users interact with directly.
- **Kernel Space:** the privileged area of memory where the operating system's kernel resides. **kernel** = core component of the operating system, responsible for managing system resources, handling hardware interactions, and providing essential services to user space. The kernel has direct access to the hardware and controls the overall operation of the computer.
- **Operating system dependency** refers to the reliance of a software application or system on specific features, functions, or behaviors of a particular operating system.



## L3 - Docker compose to launch containers

November 19, 2023 7:07 PM

[Ten simple rules for writing Dockerfiles for reproducible data science](#)

**Dockerfile:** used to build a Docker image that encapsulates an application and its dependencies. = **commands** to set up the base image, install dependencies, copy files, and configure the environment inside the Docker container.

**Create a Dockerfile:** open docker first.

1. In terminal, set current wd as the directory(folder) that contains the application(ex: app.js) you want to package.

Open VS code by: **code .**

```
$ mkdir hello-docker
mkdir: created directory 'hello-docker'

(base) NICOLE@nicoco-PC ~/OneDrive/Desktop
$ cd hello-docker

(base) NICOLE@nicoco-PC ~/OneDrive/Desktop/hello-docker
$ code .
```

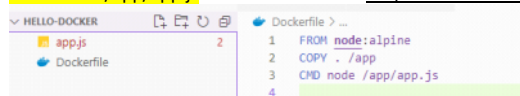
2. Open a new file in VC code, name it **Dockerfile**, in the Dockerfile:

**FROM** <base image> # <base image> =the image (with some files already) we choose to build on. Can be found on docker hub.

**COPY** ./app # copy all files in current wd into the app directory into the image so that image has a file system, and in that file system we create a directory called app.

**RUN** <command> # Run <command> during the building of image. <command> can be: apt-get update

**CMD** node /app/app.js # Run command only when create a container. Because app.js is inside the app directory, so add directory name prefix.



- In Dockerfile: Pin version number of packages, container image are necessary. Otherwise the docker will auto use the latest version, but the latest version is keep changing, which is not good for replicate your application.
- analysis data should NOT be included downloaded/installed in the container image.
- analysis code that you wrote should be included in the container image.

3. Save the Dockerfile, go to terminal: **docker build --tag <image\_name>:1.0 .**

# after --tag(or -t), specify a self-made image\_name, after <image\_name>, can add **optional Tag** by **:** (here Tag **:1.0**), followed by **absolute/path/of/Dockerfile**, if the Dockerfile is in the current wd, use **.**

```
(base) NICOLE@nicoco-PC ~/OneDrive/Desktop/hello-docker
$ docker build -t hello_dockerr .
```

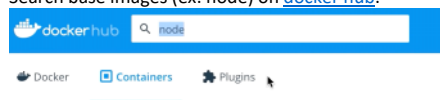
Now the image is created and stored.

**See all images on the computer**, in terminal: **docker images** or **docker image ls**

```
$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
hello_dockerr        latest             f61b6ae27810       3 minutes ago      142MB
rocker/rstudio       4.3.2             c4bfff4a95b3a      2 weeks ago        1.99GB
rocker/r-ver         4.3.2             edf6b4a63534       2 weeks ago        833MB
continuumio/miniconda3 23.9.0-0         6fbaadd54391       4 weeks ago        536MB
jupyter/minimal-notebook notebook-7.0.6     e75d34b8f32b       4 weeks ago        1.56GB
hello-world          latest            9c7a54a9a43c       6 months ago       13.3kB
```

- \* **'REPOSITORY'**: The name of the image repository.
- \* **'TAG'**: The tag associated with the image, indicating a specific version or variant.
- \* **'IMAGE ID'**: A unique identifier for the image.
- \* **'CREATED'**: The timestamp of when the image was created.
- \* **'SIZE'**: The size of the image.

Search base images (ex: node) on [docker hub](#):



**Run image to create container:** In terminal, does not matter where the current wd is: **docker run <image\_name>**

```
(base) NICOLE@nicoco-PC ~/OneDrive/Desktop/hello-docker
$ docker run hello_dockerr
Hello Docker!
```

\*If want to modify files in the image, has to rebuild the image again after modification by **docker build** with the same image name, then the image will be overwritten and you can then **docker run** the rebuilt image.

**Adding packages in existing image by:**

- 1) docker run --rm -it <image\_name> bash
- 2) In interactive terminal, use `conda install -y <package_name>` to test if it can be installed successfully.
- 3) If test is successful, open Dockerfile and add install command under **RUN**
- 4) Docker build again: `docker build --tag IMAGENAME_version2 .`

**Publish image on [docker hub](#)** so that others can pull and use it:

Assume the pushed image *hello-docker* is in the repository named *codewithmosh*



**Pull a docker image from Docker Hub repo:** in terminal, does not matter where the current wd is: **docker pull <Docker\_Hub\_repo\_name/image\_name>:version\_number**  
:version\_number is optional



```
$ docker pull codewithmosh/hello-docker
Using default tag: latest
latest: Pulling from codewithmosh/hello-docker
e95f33c60a64: Pull complete
f364f94a7409: Pull complete
cc1a7b676aaa: Pull complete
36b591a85c8e: Pull complete
a001d23672ea: Pull complete
Digest: sha256:531d3ba84f908666cd5e5062fff7d8828abe5d32c206138f399b7267a4aad462
Status: Downloaded newer image for codewithmosh/hello-docker:latest
docker.io/codewithmosh/hello-docker:latest
```

- Check if the image is pulled:

```
$ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
codewithmosh/hello-docker	latest	fd84d4d9cb44	2 years ago	112MB

- Run the pulled image with docker run <name specified in REPOSITORY>:

```
$ docker run codewithmosh/hello-docker
Hello World!
```

We can take any application and dockerize it by adding a Dockerfile to it. This dockerfile contains instructions for packaging an application into an image. Once we have an image, we can run it virtually anywhere, on any machine with docker.

### Exercise - running a container with Jupyter as a web app

Use Docker to run a container that contains the Jupyter web-application installed:

- 5) **Open docker**
- 6) **Install image in terminal:** `docker pull jupyter/minimal-notebook:notebook-7.0.6`
- 7) **Run container in terminal:** `docker run --rm jupyter/minimal-notebook:notebook-7.0.6`

--rm: automatically removes the container when it exits.

-p 8889:8888 = map **port 8888 from the container** to **port 8889 on the host machine**. The two can be the same number, as long as they are unique on the **host machine** (the physical machine on which the Docker engine is running.)

**\*any changes (install R packages in a container with RStudio as a web app) done during the runtime of the container will NOT be saved after close the container.**

Use **exit** to exit the interactive container terminal.

### To save changes made in a container:

**Mounting volumes to containers:** allows you to share data between the host machine and the container.

**Volume** = a specially designated directory within one or more containers.

- o **Persistence:** Data in volumes is preserved even if the container stops or is removed.
- o **Sharing Data Between Containers:** Volumes can be shared between multiple containers, allowing them to access and modify shared data.

**When you mount a volume:** changes made to the mounted directory in the container are reflected on the host machine, and vice versa.

**On Windows,** the laptop path depends what shell you are using:

Run in Windows terminal, then the command should be:

`docker run --rm -it -v /$(pwd):<PATH_ON_CONTAINER> <IMAGE_NAME>` to share the current directory.

Run in Power Shell:

`docker run --rm -it -v <ABSOLUTE_PATH_TO_CONTAINER>:<PATH_ON_CONTAINER> <IMAGE_NAME>`

And the path must be formatted like: `C:\Users\tiffany.timbers\Documents\project\:/home/project`

In real world, we need to create complete applications that will be using different services (**A service** = a definition for a container along with its configuration) and they need to interact with one another.

**Docker compose:** tool to define and manage multi-container Docker applications. Use `docker-compose.yml` to specifying how the containers (services) should be configured (either from same or different images) and how they interact with each other.

- Start all services with a single command: `docker compose up` = conda activate
- Stop all services with: `docker compose down`
- Scale up selected services when required.

`docker-compose -v` #show compose version. Usually installed with Docker.

```
(base) NICOLE@nicoco-PC ~/OneDrive/Desktop
$ docker-compose -v
Docker Compose version v2.20.2-desktop.1
```

`docker-compose.yml` records how the container should be launched, include which Docker image to use, ports to expose, environment variables, and more.

### Create docker compose file

Can be create at any location on your computer. Standard name = docker-compose.yml

1. **Create the file:** Open terminal in the wd where you want to create the docker compose file: `.code` to open VS code.

```
(base) NICOLE@nicoco-PC ~/OneDrive/Desktop
$ mkdir DockerComposeFile
mkdir: created directory 'DockerComposeFile'

(base) NICOLE@nicoco-PC ~/OneDrive/Desktop
$ cd DockerComposeFile/

(base) NICOLE@nicoco-PC ~/OneDrive/Desktop/DockerComposeFile
$ code .
```

In VS code, create a new file and name it **docker-compose.yml**.

2. **Specify docker-compose.yml file:**

**version: '3'** # specify the version of Docker Compose syntax

**services:** #define services

**service\_name\_1:** # self-name the service 1

**image: <image\_name>** # image to be used for service\_1

... # other optional configuration of the service\_1 like environment variables, volumes, ports, etc.

**service\_name\_2:** # self-name the service 2

**image: <image\_name>** # image to be used for service\_2

...

**\*<image\_name> here do not need to be local image, can be image from Docker Hub.**

3. **Check validity** of the file before running it: `docker-compose config`

4. **Run docker-compose.yml** file to start and manage multiple containers as a single application: `docker-compose up -d`

# -d tells Docker compose to run the container in the background, optional.

Now if use `docker ps` to see running containers, will see 2 containers (service\_name\_1 and service\_name\_2) running by specified images.

5. Stop services by: `docker-compose down`

Ex: more complex docker-compose file

services:

analysis-env:

image: rocker/rstudio:4.3.2

ports:

- "8787:8787"

volumes:

- ./home/rstudio/project

environment:

PASSWORD: password

#### **Dockerfile vs. docker-compse.yml:**

**Dockerfile** is used to define the contents and build process of a single Docker image, while the **docker-compose.yml file** is used to define and manage the configuration of multiple containers that work together as a service or application. The two often work hand-in-hand, with the Dockerfile being used to create individual images, and the docker-compose.yml orchestrating their collaboration.

# Docker commands

November 20, 2023 7:49 PM

## Docker basic commands:

`docker version` # get detailed information of the docker client and server

`docker -v` # version of docker

`docker info` # detailed info of docker installed, include number of containers/images,...

`docker --help` # get help of commands

`docker <command> --help` # get help of the specific docker command.

`docker login` # login to your docker hub account

### Images:

`docker images` # list all image in your computer

`docker pull <image_name>` # pull image from docker hub repo

`docker rmi <image_ID>` # remove one or more images, <image\_ID> can be found by **docker images**

### Containers:

`docker ps -a` # list all containers (running version of image) both running and exit, on your system.

`docker run <image_name>` # run a container, if cannot find it locally, will search on Docker Hub and download.

`docker run -it <image_name>` # interactively start the image and login to it = start the command prompt of the image

`docker run --rm -it <image_name> bash` # Run the image in a bash terminal instead of the web app

`docker start <container_ID>` # start the container. <container\_ID> can be found by **docker ps**

`docker stop <container_ID>` # stop the container. Container will be in **exit** status.

`docker pause <container_ID>` # pause the container

`docker unpause <container_ID>` # unpause the container

`Docker attach <container_ID>` # attach local standard input, output, and error streams to a running container

### System:

`docker stats` # return details on running container's memory usage, input/output, etc.

`docker system df` # check disk usage of docker

`docker system prune` # remove unused data. **Be careful to use this command.**

# L4 - Introduction to testing code for data science

November 22, 2023 4:20 AM

<https://ubc-dsci.github.io/reproducible-and-trustworthy-workflows-for-data-science/materials/lectures/06-intro-to-testing-code.html#common-types-of-test-levels-in-data-science>

Write test before writing functions.

Workflow for writing functions and tests for data science:

1. **Write the function specifications and documentation** - but do not implement the function. This means that you will have an empty function, that specifies and documents what the name of the function is, what arguments it takes, and what it returns.  
In R, to get the outline of the function documentation: `code -- insert roxygen skeleton`
2. **Plan the test cases and document them.** Your tests should assess whether the function works as expected when given correct inputs, and that it behaves as expected when given incorrect inputs (e.g., throws an error when the wrong type is given for an argument). For the cases of correct inputs, test the top, middle and bottom range of these, and all possible combinations of argument inputs possible. Also, the test data should be as simple and tractable as possible.
3. **Create test data used for testing your function.** In data science, you likely need to create both the data that you would provide as inputs to your function, as well as the data that you would expect your function to return.
4. **Write the tests** to evaluate your function based on the planned test cases and test data.
5. **Implement the function** by writing the needed code in the function body to pass the tests.
6. **Iterate between steps 2-5** to improve the test coverage and function.

**Common types of test levels** in data science

**Unit tests** - exercise individual components, usually methods or functions, in isolation.

- quick to write and the tests incur low maintenance effort since they touch such small parts of the system.
- It ensure that the unit fulfills its contract making test failures more straightforward to understand.
- Ex: test for count\_classes

**Integration tests** - exercise groups of components to ensure that their contained units interact correctly together.

- touch much larger pieces of the system and are more prone to spurious failure. Since these tests validate many different units in concert, identifying the root-cause of a specific failure can be difficult.
- In data science, this might be testing whether several functions that call each other, or run in sequence, work as expected
- Ex: tests for a tidymodel's workflow function

**Test-driven development (TDD):** a process for developing software: write test cases that assess the software requirements **before** write the software that implements the software requirements/functionality.

**Testability** = the degree to which a system or component facilitates the establishment of test objectives and the execution of tests to determine whether those objectives have been achieved.

**Properties of effective test writing and execution:**

- **controllability:** the code under test needs to be able to be programmatically controlled
- **observability:** the outcome of the code under test needs to be able to be verified
- **isolateability:** the code under test needs to be able to be validated on its own
- **automatability:** the tests should be able to be executed automatically.

Ex: in your git\_repo

Git\_repo/tests:

Testthat.R:

```
Library(testthat)  
Library(demo-test-ds-workflow)  
testthat folder
```

```
Git_repo/tests/testthat:  
helper-count\_classes.R :  
test-count\_classes.R
```