

### **Interactive Lectures**

#### All lectures in the course will be interactive

They contain running code, as well as theory!

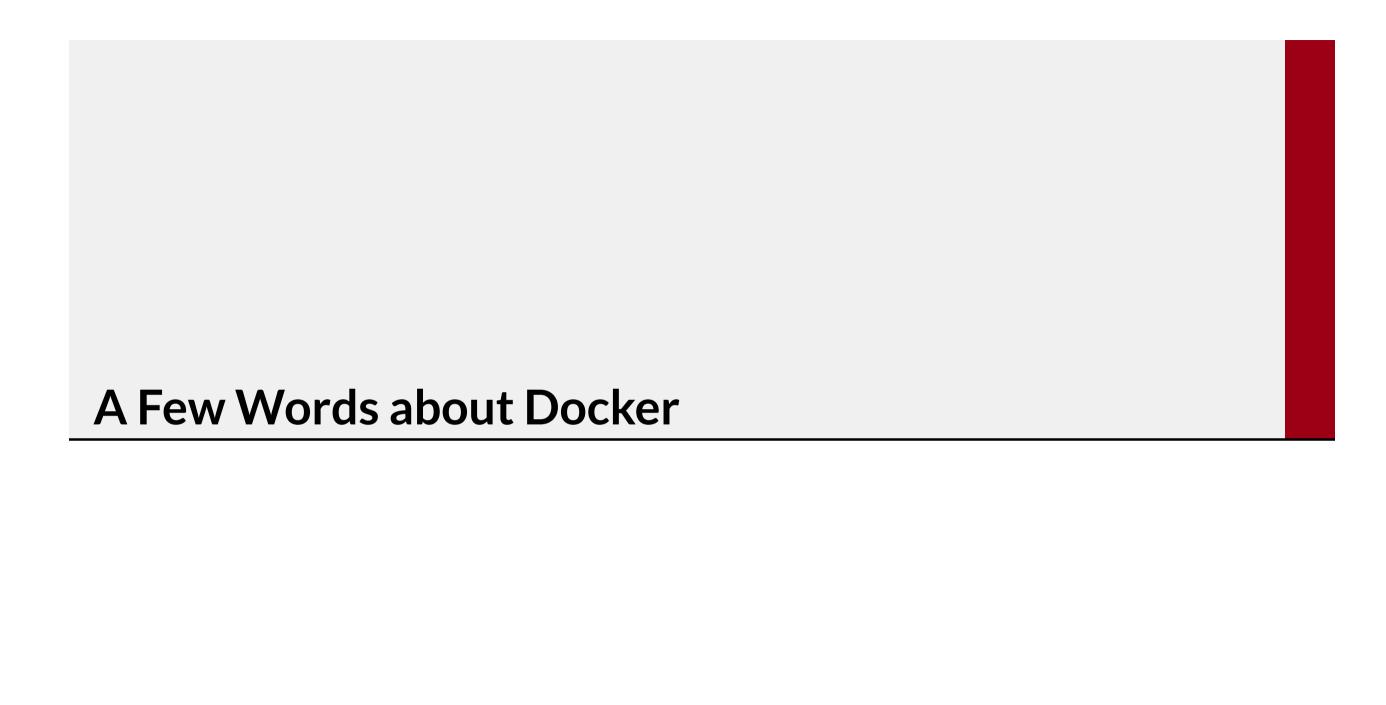
- Presented and discussed in frontal lectures...
- ...You can download PDFs
- ...But you will also be able to make changes and experiment

### From a software perspective, the workshorses of this approach are:

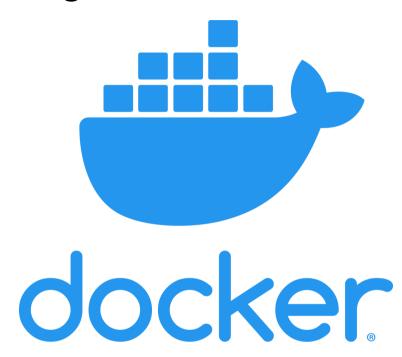
- Jupyter notebooks for the presentation & interaction
- <u>Docker</u> containers for the setup and distribution

### Both are widely used systems:

- Jupyter is a user favorite when it comes to data science
- Docker is a state-of-the-art system for manageing services



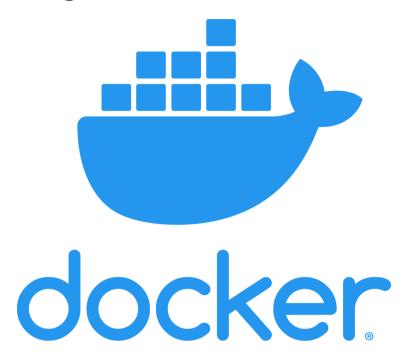
Docker is a system for running software in "containers"



Think of a container as a lightweight virtual machine:

- (Essentially) the same level of isolation
- ...But smaller disk footprint, faster setup and operation, etc.

### Docker is a system for running software in "containers"



Using containers has many advantages:

- Multiple environments on the same machine
- Improved isolation, robustness, and reproducibility
- Easier replication
- Scalability of cloud services...

### During this course we will see many problems

...And tackle them with many techniques:

- Classical Machine Learning
- Deep Learning
- Statistics
- Signal processing
- Declarative optimization
- Differential Equations
- **..**

Managing dependences can become hellish

### With docker, we can simply use a different container per case study

Inside each container we will have:

- All the needed libraries & tools
- A running instance of a Jupyter server

In the host machine (your PC):

- We will just open a browser...
- ...And connect to the Jupyter server

### Two key concepts in Docker

- A container is a (sort of) running, lightweight, Virtual Machine
- An image is (sort of) the content of the hard disk of the VM

The image can be used to instantiate multiple containers

# **Building an Image**

### Images in docker are built by:

- Starting from a base image on <u>Docker Hub</u>
- Copying content between the host and the container
- Running commands in the container

### The process is controlled via a Dockerfile

- Just a text file with a specific syntax
- There is an <u>extensive reference</u>, but we only care about a few commands

### To build an image, we can use:

docker build .

...From the directory with the Dockerfile

### This is a simple Dockerfile example:

```
FROM python:3.8
RUN pip install jupyter pandas sklearn matplotlib ipympl RISE
COPY . /app
WORKDIR /app/notebooks
CMD ["jupyter", "notebook", "--port=8888", "--no-browser", "--ip=0.0.0.0", "--allow-root"
```

■ The FROM keyword specifies the base image

```
FROM python:3.8
RUN pip install jupyter pandas sklearn matplotlib ipympl RISE
COPY . /app
WORKDIR /app/notebooks
CMD ["jupyter", "notebook", "--port=8888", "--no-browser", "--ip=0.0.0.0.0", "--allow-root"
```

- The RUN keyword runs a command
- In our case, we install a number of python packages

```
FROM python:3.8
RUN pip install jupyter pandas sklearn matplotlib ipympl RISE
COPY . /app
WORKDIR /app/notebooks
CMD ["jupyter", "notebook", "--port=8888", "--no-browser", "--ip=0.0.0.0", "--allow-root
```

- The COPY keyword transfers data from the host to the container
- The first path refers to the host
- The second path to the container

```
FROM python:3.8
RUN pip install jupyter pandas sklearn matplotlib ipympl RISE
COPY . /app
WORKDIR /app/notebooks
CMD ["jupyter", "notebook", "--port=8888", "--no-browser", "--ip=0.0.0.0", "--allow-root
```

- The WORKDIR changes the current directory in the container
- It's like running cd in the container

```
FROM python:3.8
RUN pip install jupyter pandas sklearn matplotlib ipympl RISE
COPY . /app
WORKDIR /app/notebooks
CMD ["jupyter", "notebook", "--port=8888", "--no-browser", "--ip=0.0.0.0", "--allow-root
```

- The CMD keyword is triggered only when we run a container
- It's the first command that the container should execute
- It does nothing when building an image

#### When we run docker build . for our file:

- The docker daemon downloads the base image, if not already available
- A container is started
- All operations in the Dockerfile are executed
- The resulting container is dumped, to create an image

### You can check that a new image has been built using:

```
docker image ls
```

You will see an entry with no name:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
<none></none>	<none></none>	96b910c1514f	3 seconds ago	1.36GB

### You can remove an image with:

```
docker image rm <image name or id>
```

- Useful to free space, however...
- ...Images are incremental! Docker stores only the differences
- ...So, don't worry too much about space usage

### You can remove all images with no running container with:

docker image prune

# **Running a Container**

#### You can instantiate and run a container with:

docker run <image name or id>

- The container stdout will be piped (i.e. connected) to your terminal
- By default, this is not the case for stdin
- You can make the container interactive with the -it options
- You can autoremove the container at the end with --rm
- You can sync folders in the host and on the container using <u>volumes</u>

The <u>documentation</u> is extensive

# Running a Container

#### You can obtain the list of all containers with:

docker ps

■ The option -a shows all containers (incl. those that are stopped)

#### You can remove a container with

docker rm <container id>

- As you see, it's a very flexible system
- ...But also a bit complex

That's why we will automate most operations using Docker Compose

# **Docker Compose**

### **Docker Compose** is a tool to help the management of containers

In a second docker-compose.yml file, you specify:

- Which "services" (i.e. container) should be built and run
- How to build them
- Which options to use when running them
- **...**

All in a <u>human-readable</u>, <u>declarative format</u>

# A Docker Compose Example

### Let's see a simple docker-compose.yml for one lecture:

```
version: '2.0'
services:
   jupyter:
   build: .
   ports:
        - "8888:8888"
   volumes:
        - .:/app
```

- version refers to the Docker Compose syntax
- services is followed by a list of the containers
- jupyter is our service
- build specifies where the Dockerfile can be found

# A Docker Compose Example

### Let's see a simple docker-compose.yml for one lecture:

```
version: '2.0'
services:
  jupyter:
  build: .
  ports:
    - "8888:8888"
  volumes:
    - .:/app
```

- ports tells which ports to expose to docker run
- volumes specifies which folders to sync
- In our case "/app/notebooks" on the container
- ...Will actually be "./notebooks" on the host

# **Benefits of Using Docker Compose**

#### We need to use one more tool, but now we can:

Build and run a container with:

docker compose up

■ The command can also restart a stopped container

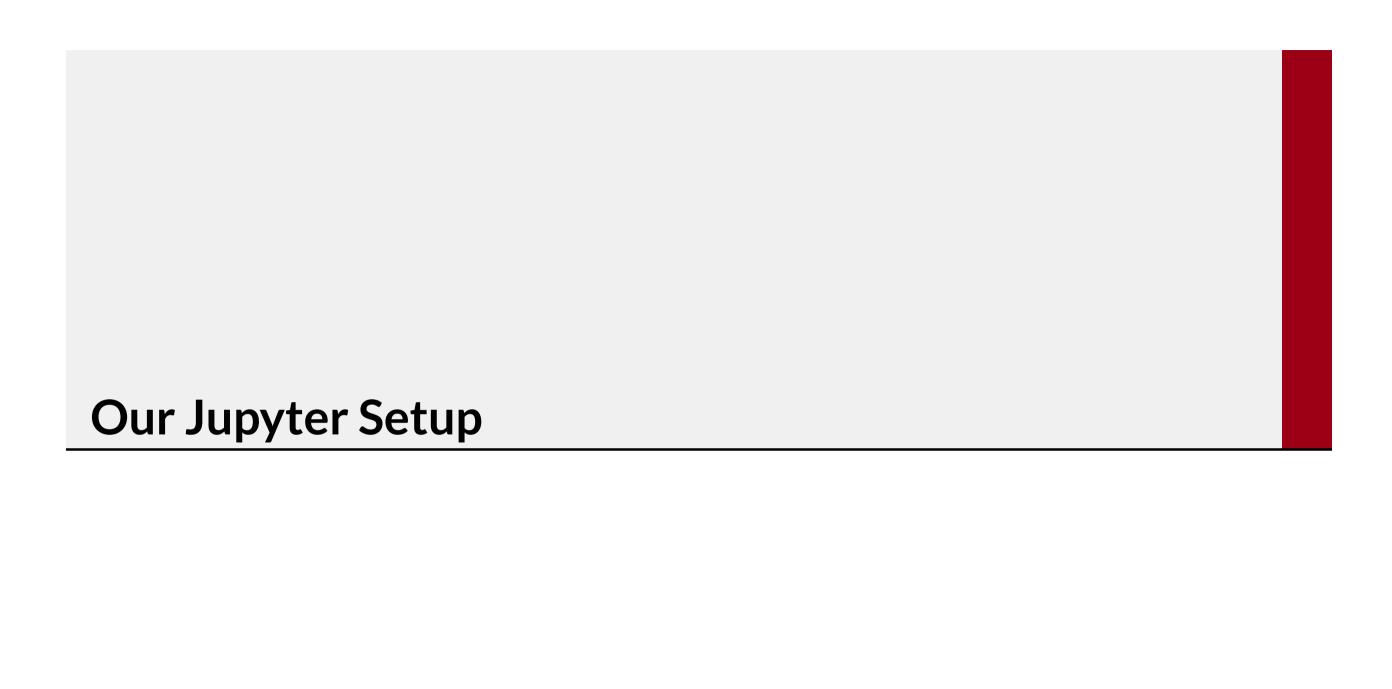
Stop the container with CTRL+C, or with:

docker compose stop

Stop and remove the container with:

docker compose down

...Which is considerably simpler than before!



### We will often work with this development setup

The folder with the notebooks is structured as follows:

```
notebook1.pynb
notebook2.pynb
...
util <-- module
assets <-- images and such
rise.css <-- for the "slide" mode</pre>
```

### We will often work with this development setup

The folder with the notebooks is structured as follows:

The most important part: we'll use modules besides notebooks

### Working with modules provides some advantages:

We do not need to keep all our code in the notebooks. We can:

- Share functions between cells
- Share functions between notebooks
- IDEs can offer more functionality if they recognize a module

### ...But also a significant disadvantage:

- Python modules are compiled first when loaded...
- ...The loaded version is not updated when the source changes

This is very inconvenient at development time

### We can circumvent this thanks to Jupyter "magic" extensions

The first one is the "autoreload" extension

```
In [1]: %load_ext autoreload
%autoreload 2
```

- load\_ext will enable the extension
- autoreload 2 will reload all modules before code execution

### This is inefficient, but convenient during development

- Together with the use of volumes (in docker-compose)...
- ...This allows us to update the code without re-building the docker image

# Starting a Notebook

Let's look back to the CMD keyword in our Dockerfile:

```
CMD ["jupyter", "notebook", "--port=8888", "--no-browser", \
    "--ip=0.0.0.0", "--allow-root"]
```

This is translated to:

```
jupyter notebook --port=8888 --no-browser --ip=0.0.0.0 --allow-root
```

- --port 8888: the server listen on port 8888
- --no-browser: do not open the browser (there's no browser in the container)
- --ip=0.0.0.0:listen on all network interfaces
- --allow-root: we operate as root (admin) on the container

# Starting a Notebook

When we run:

```
docker compose up
```

The output will look like:

```
Starting ad_stat_jupyter_1 ... done
...
...Use Control-C to stop this server and shut down all kernels...
...
...
...To access the notebook...
...copy and paste one of these URLs:
... http://34b908cf2362:8888/?token=82e337a2be9915cdebce276bf...
... or http://127.0.0.1:8888/?token=82e337a2be9915cdebce276bf...
```

■ The last URL can be copy-pasted in your favorite browser

# Starting a Notebook

When we run:

```
docker compose up
```

The output will look like:

```
Starting ad_stat_jupyter_1 ... done
...
...Use Control-C to stop this server and shut down all kernels...
...
...
...To access the notebook...
...copy and paste one of these URLs:
... http://34b908cf2362:8888/?token=82e337a2be9915cdebce276bf...
... or http://127.0.0.1:8888/?token=82e337a2be9915cdebce276bf...
```

■ The token is cached by the browser and grants access to the notebooks

-1.0

- We will somtimes use the ipympl package and the widget jupyter magic
- This will display basic tools to rescale and zoom images

```
In [2]: #%matplotlib widget
        from matplotlib import pyplot as plt
        import numpy as np
        x = np.linspace(0, 2*np.pi, 100)
        plt.figure(figsize=(9, 3))
        plt.plot(x, np.sin(x))
        plt.tight_layout()
          1.0
          0.5
          0.0
          -0.5
```