The first docker commands

```
docker run -it ubuntu
```

Here's a breakdown of what each part of this command does:

docker run: This is the command used to create and start a new container from a specified image. -it: These are two flags combined: -i (interactive): Keeps the STDIN (standard input) open even if not attached. -t (tty): Allocates a pseudo-TTY (terminal), which allows you to interact with the container via the terminal. ubuntu: This is the name of the Docker image you want to run. In this case, it is the official Ubuntu image from Docker Hub. When you run this command, Docker will:

Pull the ubuntu image from Docker Hub if it is not already available locally. Create a new container from the ubuntu image. Start the container and attach your terminal to it, allowing you to interact with the container's shell. This command is useful for starting a container where you need to interact with the shell, for example, to run commands or scripts manually.

Once you've started the linux ubuntu container, a shell opens up with "root@...". You could update the linux version of the container with linux command `apt-get-update`. If you want to install a programm with name "tree" for example you need the linux command `apt-get install tree`

You could do the same to run python:

```
docker run -it python
```

Above installs the latest python version from docker hub. If you need a different version, e.g. 3.8 then do the following:

```
docker run -it python:3.8
```

Docker Image Tag

The `docker image tag` command is used to create a new tag for an existing Docker image. This command does not alter the image itself but allows you to refer to the same image by a different name or tag within a Docker registry or on your local system.

The general syntax for the command is:

- `SOURCE_IMAGE[:TAG]` refers to the existing image and optional specific tag you want to add a new tag to. If the tag isn't specified, Docker assumes `latest`.
- `TARGET_IMAGE[:TAG]` refers to the new name and tag you want to assign to the image.

This is particularly useful for versioning images or when you need to push them to a different repository.

Example:

```
docker image tag my-image:1.0 my-image:latest
```

Docker Image Remove

The `docker image remove` (or `docker rmi`) command is used to delete one or more Docker images from your local Docker environment. This helps in managing disk space and keeping your local environment clean.

The general syntax for the command is:

```
docker image remove [OPTIONS] IMAGE [IMAGE...]
```

or its shorthand:

```
docker rmi [OPTIONS] IMAGE [IMAGE...]
```

Here, `IMAGE` can be the image ID, image tag, or the image name with a specific tag.

Some important points about `docker image remove`:

- An image cannot be removed if there are any containers using it, unless you force the removal.
- If you want to remove all unused images, you can combine it with commands like `docker image prune`.

Example of removing an image:

```
docker image remove my-image:latest
```

This will remove the image tagged `my-image:latest` from your system if no containers are currently using it. You can use the `-f` or `--force` option to forcibly remove an image being used by stopped containers.

Docker Run vs. Docker Start and Docker Create

The `docker run`, `docker create`, and `docker start` commands are used to manage the lifecycle of Docker containers, but they serve different purposes in that process.

`docker run`

- `docker run` is the most commonly used command for running containers.
- It combines the functions of `docker create` and `docker start`.
- When you run `docker run`, Docker does the following:
  1. **Creates** a new container from the specified image.
  2. **Starts** the container.
  3. **Attaches** to the container's output (unless `-d` or `--detach` is specified to run the container in the background).

- It is equivalent to running `docker create` followed by `docker start`.
- Example:

```
docker run ubuntu:latest
```

`docker create`

- `docker create` is used to create a new container without starting it.
- This command is useful if you want to set up a container's configuration before starting it, such as setting environment variables or network settings.
- It prepares the container and returns the container ID without actually starting it.
- Example:

```
docker create --name my_container ubuntu:latest
```

`docker start`

- `docker start` is used to start an existing, stopped container.
- If you use `docker create` to prepare a container, you'll use `docker start` to start it whenever needed.
- It only starts containers and does not create new instances.
- It does not attach to the container's output by default, which means it runs the container in the background (to attach, use `docker start -i`).
- Example:

```
docker start my_container
```

Differences:

- `docker run` combines creation and starting of a new container and is a one-step command to both create and start with output attached to it
- `docker create` only creates the container, allowing for delayed starting or further configuration beforehand.
- `docker start` only works on existing containers that have been created, making it suitable for restarting purposes.

Run a new container

```
docker run ubuntu:latest
```

Run a new container but keep it running in the background (-d flag for detached)

```
docker run -d ubuntu:latest
```

This is similar to a combination of `docker create`& `docker start`

Create a container without starting it

```
docker create --name my_container ubuntu:latest
```

Start the previously created container

```
docker start my_container
```

Docker Inspect, Docker History, Docker Logs

Above commands are utilities for obtaining detailed information about Docker images and containers. Each serves a unique purpose in providing insight into Docker objects.

`docker inspect`

- `docker inspect` is used to return detailed information about Docker containers, images, volumes, or networks.
- It provides comprehensive JSON output with all the configurations and metadata of the specified object.
- Commonly used to troubleshoot or understand the configurations such as environment variables, mounts, network settings, etc.
- You can filter specific data using the `--format` option.
- Example:

```
docker inspect my_container
```

`docker history`

- `docker history` displays the history of an image, showing the layers, and the creation history.
- It lists all layers that make up an image with details like command, size, and timestamp for each layer.
- Useful for understanding how an image was built and what commands were used to create each layer.
- Example:

```
docker history my_container
```

‘docker logs“

This command retrieves the logs from a running or stopped container. It shows the standard output and standard error streams from the container’s main process. For example, `docker logs <container_id>` will display the logs output by the application running in the specified container.

```
docker logs my_container
```

Docker Rename and the flag –Name

The `docker rename` command and the `--name` flag are both used to manage container naming but in different contexts.

`docker rename`

- The `docker rename` command is used to change the name of an existing Docker container.
- This is useful if the original name assigned to the container is not descriptive or was given a generic name but you later wish to rename it for clarity.
- The syntax for renaming a container is:
- Example: To rename a container named `my-old-container` to `my-new-container`, you would run:

```
docker rename my-old-container my-new-container
```

`--name` flag

- The `--name` flag is used during the container creation process (e.g., with `docker run` or `docker create`) to assign a specific name to a container.
- By default, Docker assigns a random and unique name to containers unless specified.
- Using `--name`, you can specify a human-readable and easily recognizable name for your container upon creation.
- Example: To run a container with a specific name, you could use:

```
docker rename my-old-container my-new-container
```

Differences:

- The `docker rename` command changes the name of an already existing container.
- The `--name` flag sets the name of a container at the time of its creation.

Rename an existing container

```
docker rename my-old-container my-new-container
```

Run a new container with a specific name

```
docker run --name my-container ubuntu:latest
```

Installation of Programs into Existing Container

To install a program into an existing container using `docker exec`, follow these steps:

1. Use `docker exec` to run a command inside the running container. For example, to install `tree` on an Ubuntu container:

```
docker exec -it [CONTAINER ID] apt-get update
docker exec -it [CONTAINER ID] apt-get install -y tree
```

2. Verify the installation by running the installed program:

   ```
   docker exec -it [CONTAINER ID] tree --version
   ```

Replace `[CONTAINER ID]` with the actual ID of your container. This process allows you to execute commands inside the container without needing to start an interactive shell session.

Above will not work for a python container. To install a package like pandas into an existing Python container using `docker exec`, follow these steps:

1. Open an new bash terminal
2. Connect new bash terminal to the existing container with

   ```
   root@container_id: docker exec -it container_id_or_name bash
   ```

3. Install pandas package into python container with

   ```
   root@container_id: pip install pandas
   ```

4. Go back to python shell:

   ```
   >>>import pandas as pd
   ```

5. Verify the installation by running a Python command to check the installed library:

   ```
   >>> print(pd.__version__)
   ```

Ports freigeben, Container im Hintergrund laufen lassen

Standardmäßig wird ein Container nicht neu gestartet. Es gibt aber eine **restart policy**

- `on-failur[:max-retries]`: Startet den Container "may-retries"-mal neu, wenn es einen Fehler gibt
- `always`: Startet den Container immer neu. Außer wir haben ihn gestoppt. Dann wird erst beim nächsten Docker Daemon erneut gestartet
- `unless-stopped`: Startet den Container immer neu (und wenn wir ihn stoppen bleibt er gestoppt)

Beispiele: Wir setzen die Restart Policy so: (Der Container läuft im Backround Modus ohne Terminal (-d) und wird dann automatisch neu gestartet)

```
docker run -d --restart unless-stopped[Image-Name]
```

Die Restart Policy kann dann so aktualisiert werden

```
docker update --restart always [Container-ID / Name]
```

oder so:

```
docker update --restart onfailure:5 [Container-ID / Name]
```

Portweiterleitung

Beim erstellen des Containers die Portweiterleitung spezifizieren, da ja der Start eines Containers auch automatisch durchgeführt werden kann.

```
docker container create -p Host-Port:Container Port Image-Name
```

Datemnagement in Docker Containern

Dateien in und aus einen Container kopieren

Entweder einen neuen Docker container starten/erstellen mit:

```
docker run ubuntu -t
```

oder einen bestehenden Container starten mit:

```
container start [container_id]
container exect it [container_id] /bin/bash
```

Dann in einer neuen Shell in diesen Container wechseln und in diesem Container dann ein neues Verzeichnis "Folder" erstellen

```
mkdir Folder
```

und dann in der urpsprünglichen Shell mit dem Befehl z.B. die Text Datei file.txt im Verzeichnis "/Users/olafstinnen/Projects/mylibrary/"

```
docker cp /Users/olafstinnen/Projects/mylibrary/file.txt [container_id]:/Folder
```

hineinkopieren.

Wechsel man dann wieder zurück in die Shell für den den Container findent man dann die Dater über "cd Folder" und dann mit "ls" wieder.

Das Verlinken von Host-Verzeichniss mit einem Container

Das Verlinken bzw. das Verküpfen einer Verzeichnisstruktur eines Containers mit Daten, die außerhalb vom Container liegen. Diese Daten, die außerhalb vom Container liegen sind persistent. Sie werden als nicht gelöscht wenn wir den Container löschen. Im wesentlichen gibt es zwei Arten von mounts: "bind" und "volume".

Bind Mounts

Bind mounts allow you to mount a file or directory from the host machine into the container. This means that changes made to the files in the container are reflected on the host and vice versa. Bind mounts are useful when you need to share data between the host and the container.

Example:

```
docker run -d \
  --name my_container \
  --mount type=bind,source=/path/on/host,target=/path/in/container \
  my_image
```

In this example:

- `type=bind` specifies that this is a bind mount.
- `source=/path/on/host` is the path on the host machine.
- `target=/path/in/container` is the path inside the container where the host path will be mounted.

Volumes

Volumes are managed by Docker and are stored in a part of the host filesystem which is managed by Docker (`/var/lib/docker/volumes/` on Linux). Volumes are useful for persisting data beyond the lifecycle of a container and for sharing data between multiple containers.

Example

```
docker run -d \
  --name my_container \
  --mount type=volume,source=my_volume,target=/path/in/container \
  my_image
```

In this example:

- `type=volume` specifies that this is a volume mount.
- `source=my_volume` is the name of the volume.
- `target=/path/in/container` is the path inside the container where the volume will be mounted.

Key Differences

1. **Location**:
   - **Bind Mounts**: Use any location on the host filesystem.
   - **Volumes**: Managed by Docker and stored in Docker's storage area.
2. **Use Case**:
   - **Bind Mounts**: Ideal for sharing data between the host and container, especially during development.
   - **Volumes**: Ideal for persisting data and sharing data between multiple containers.
3. **Management**:
   - **Bind Mounts**: Managed by the host system.
   - **Volumes**: Managed by Docker, which provides better isolation and management.

Example Code for Both

```
# Bind Mount Example
docker run -d -it \
  --name my_bind_container \
  --mount type=bind,source=/path/on/host,target=/path/in/container \
  my_image
```

With "pwd" you get the source path e.g. "/Users/olafstinnen/Projects/mylibrary" and should in destination "Projects" in container and my_image is Ubuntu

```
docker run -d -it \
  --name Olafs_Container \
  --mount type=bind,source=/Users/olafstinnen/Projects/mylibrary,target=/projects \
  ubuntu
```

output is this container in docker hub

and looking into the container the "projects" folder is there

Volume Mount Example

Same result from above but with volume mount instead of bind mount and as a short command:

```
docker run --name="Olafs_Container" -d -it -v$(pwd):/project ubuntu
```

Troubleshooting

If you enter this command

```
docker run -it ubuntu
```

and feedback is this docker: Cannot connect to the Docker daemon at unix:///Users/olafstinnen/.docker/run/docker.sock. Is the docker daemon running?

Then you need to start docker desktop via Launchpad or via terminal command `docker desktop start`

Infos from Container