Docker

What Is Docker?

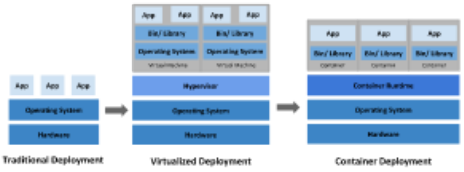
> Docker containerises a program so it will run on any environment by bundling an application and all of its dependencies into a container. It even includes the operating system (OS).

- Standardised development environment across many workstations
- Consistent deployment of applications
- Full reproducibility (e.g. for research)

> A container can be considered a virtual machine (VM); however, whilst a VM virtualises the hardware (the available RAM), a container only virtualises the OS. Note that Docker does not simply make a copy of the OS you want to work with; rather, it provides the necessary tools for working with a specific OS.

> If an application runs in the latest ubuntu version, Docker will not install the latest version each time the application is run. It will simply obtain the tools necessary to run that version without installing an entire OS.

> Furthermore, although the tools need to be installed, Docker installs them only once (unless they are uninstalled) and accesses them the next time the same application is run. To do so, Docker relies on a powerful tool named Docker images, which are the templates for running containers.



> Conventionally, a regular application is quite lightweight during deployment. This is because only the code required to run the application is deployed; however, we are not usually aware of the OS on which the application will be run. To deploy the entire VM, for example, we must install and launch the entire OS every time the application is run. As a solution, containers offer both advantages of the lightweight deployment

> without needing to install the OS for each instance of the software and the ability to run the application consistently in multiple environments.

- Containers are lighter than VMs.
- They are **immutable**, meaning that their contents remain constant.
- A container can easily be created or destroyed depending on the user's needs.
- All the dependencies of an application can be packed in a container, including the OS.
- Containers are reproducible since they run consistently, regardless of the host OS.
- **Micro-services architecture** is supported. An app can be divided into multiple separate containers that communicate with each other:
  - Only the required image is changed.
  - There is no need to deploy everything anew.
  - The components are easily switchable.

Docker Image

> A Docker *image* is a blueprint that indicates all the steps required to run the container that holds an application. Initially, when the image is `built`, the required tools are downloaded and installed. Subsequently, Docker accesses the tools when the application is run again.

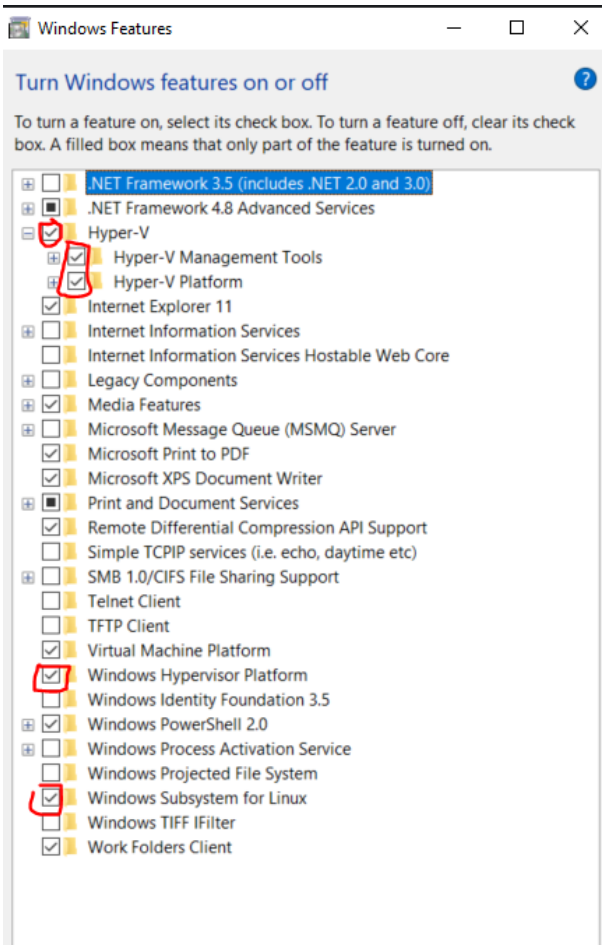
> To ensure efficiency, Docker shares these tools between images; thus, if another image is created where these tools are used again, Docker will not download or install them because it is aware of their location.

> Docker enables us to package code, apps, etc. with all the necessary dependencies in a self-contained environment called a `Docker image`. Afterward, we can create instances of these images called `Docker containers`

Set Up

> When creating a Docker image, your computer must create containers that can allocate a `memory slot` for running certain tools (even though they are not VMs) that your OS might not support.

> To grant access to these memory slots, your computer must start an engine that creates the containers within your computer. Different OSs have different approaches for carrying out this task. For example, for Windows and iOS, Docker Desktop must be installed, which will handle the creation of the corresponding engine that will create the containers. Conversely, Linux distributions are considerably more flexible, and Docker can be run simply by installing the engine that will create the containers.



1. https://docs.docker.com/desktop/
2. Download for your OS
3. To ensure that the installation was successful, run `docker --version` in the CLI (This applies to all OSs).
4. In the Windows search bar, type 'Turn Windows Features On or Off' and subsequently **enable** the following
5. The latest version of WSL might also be required

Docker Hub

> Docker Hub is a repository for storing Docker images created by users globally.

> Bear in mind that you will utilise base images to create Docker images, and these base images will be stored in Docker Hub.

1. https://hub.docker.com/

2. Make an account

Docker Images + Dockerfile

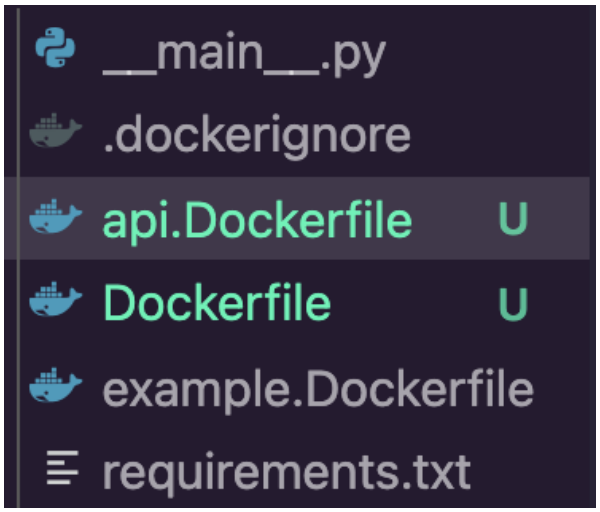
> Docker images are the instructions required to create an instance of a Docker container.

> Thus, Docker images are essentially a set of steps followed by the Docker engine to create the environment for running an application. These steps are declared in a file named `Dockerfile`. Docker searches for this special file whenever an image is to be built.

> `Dockerfiles` do not contain any extensions. The name of the file is literally `Dockerfile`; however, an extension may be used. For example, if the Dockerfile specifies the steps for creating an image for an API image, it can be called `api.Dockerfile`.

> The smaller the image, the better

> Use official images if possible



When a Dockerfile is created in VSCode, it will automatically be recognised as a Dockerfile, as indicated by the characteristic whale icon.

Once the Dockerfile is created, you can start containerising your application. However, you need to specify the commands you want Docker to run.

> Dockerfiles will contain instructions, such as `FROM`, `RUN`, `CMD`, `COPY`, ... The uppercase words that start each line in the Dockerfile are called **instructions** and are basically commands, followed by arguments (similar to the case in the terminal), which the docker build command knows how to execute. Docker build runs each of these instructions in turn to create the docker image.

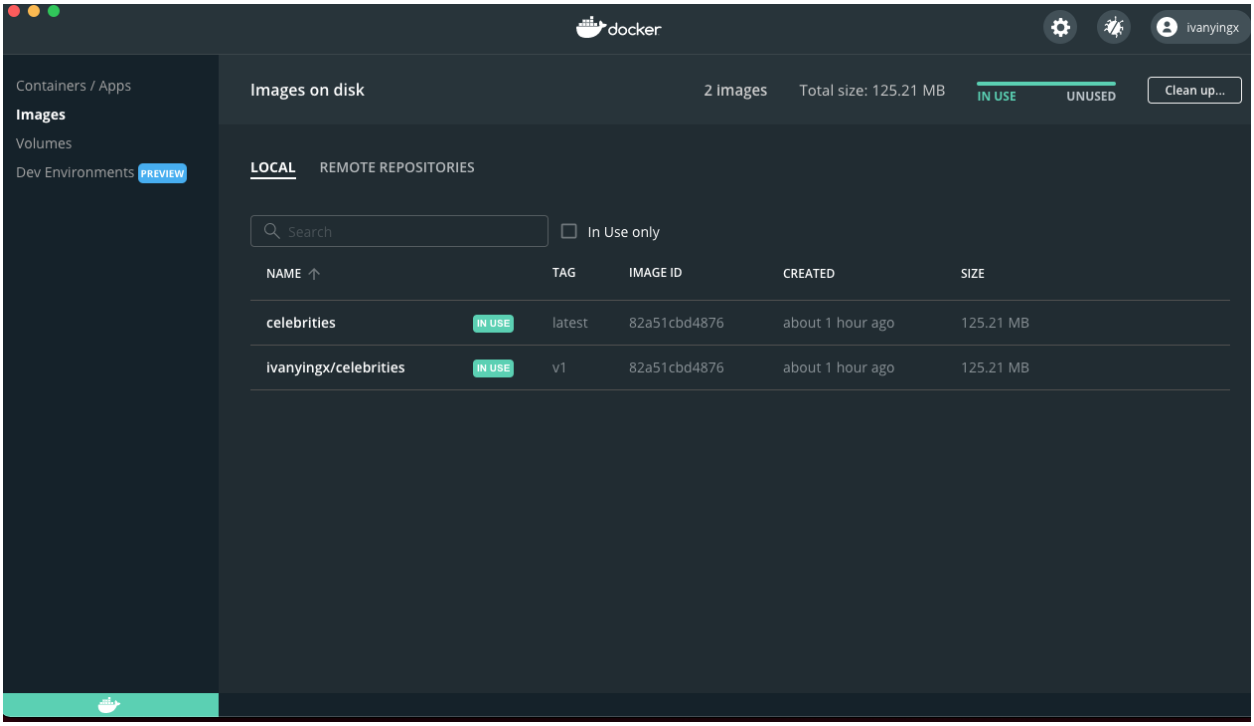
```
FROM python:3.8-slim-buster
COPY . .
RUN pip install -r requirements.txt
CMD ["python", "scraper/celebrity_scraper.py"]
```

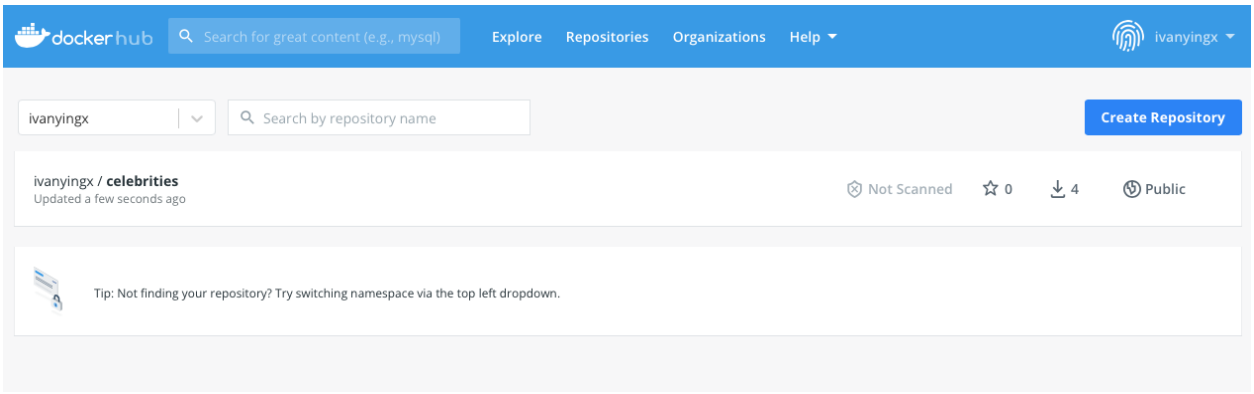
> Temporary files are left out and increase image's size `rm -rf /var/lib/apt/lists /*`

1. `cd` to the folder
2. Make a Dockerfile
3. Write the file
4. Conventionally, Docker images are built from a pre-built image Docker that can be found on Docker Hub. The pre-built image usually contains some dependencies. A common practice is to use an image with Python installed. You can download and run the pre-built image using the `FROM` clause
5. Install or copy anything of our choosing into the docker container. Remember that the directory you add is relative to the location of the Dockerfile.
6. This will copy everything in the Dockerfile directory (`requirements.txt` and the `scraper` folder) into the container.
7. Understanding this step is extremely important. When an image is built, the relevant files are copied into the container, which is analogous to copying them into a different and separate computer. In other words, it is almost as if there is a separate mini-computer containing the scraper, with Python installed.
8. The first `.` argument following the `COPY` instruction is the location of the assets *on your machine* that you wish to copy. The second `.` argument following the `COPY` instruction is the location where the assets will be copied to *on the Docker container*.
9. As the final step before running the scraper, your python packages must be installed, e.g. `beautifulsoup` and `requests`. Fortunately, the requirements file was also copied into the image; thus, the packages can be installed directly using the `RUN` command, which runs the bash command that it follows.
10. Now, we run the python script. Note that the `RUN` clause is unsuitable here because `RUN` is executed when the image is built. We need a command that is executed when the image is run, and that clause is `CMD`.
11. The `CMD` clause can be declared in many ways. In this case, we employ square brackets, and the first item is the executable (`python`), while the rest are the parameters (files).
12. Build the image. In the CLI, change the directory to `celebrity_example` . Thereafter, use the `build` command from Docker `docker build [OPTIONS] [Dockerfile path]`
13. The typical naming convention for Docker images is `image_name:version`. Typically we specify the version as `latest` rather than manually writing out the semantic versioning label.
14. Since we are in the same directory as the Dockerfile, the Dockerfile path is simply a dot (`.`).
15. Run the following command in your command line to build the container: `docker build -t celebrities:latest .`
16. Now, we verify if the image was successfully created by running the following command: `docker images`

17. Run it `docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]`
18. Run the image `docker run celebrities` This will throw an error because the script expects an input. However, at present, this is impossible because the image is running in a non-interactive mode. As a solution, we must add the options, `-i` and `-t`: `-i` will keep the STDIN open, and `-t` will make the process interactive.
19. Run `docker run -it celebrities` The image can be used everywhere, regardless of the OS and dependencies installed.
20. Additionally, you can distribute it globally using Docker Hub. To do this, login to the Docker Hub account by running `docker login` and enter your credentials.
21. The images you push to Docker hub need to have a specific name `<username>/<image_name>:<tag>` create a copy of the existing image with a new name using the docker tag command. The Image_id can be observed when `docker image` is run `docker tag 82a51cbd4876 ivanyingx/celebrities:v1` 'ivanyingx' is the username, which you should replace with yours, and 82a51cbd4876 is the Image_id.
22. Confirm that the image has been properly built by running docker images once more. You can also confirm it in Docker Desktop
23. Finally, we push the image to Docker Hub. Pushing an image to Docker hub is similar to pushing a repository to GitHub. Simply use docker push `docker push ivanyingx/celebrities:v1`
24. To verify that your image has been uploaded, go to your Docker hub account

```
docker build [OPTIONS] [Dockerfile path]
docker build -t celebrities:latest

docker images # show our current images on this machine

docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]
docker run celebrities
docker run -it celebrities

docker login

<username>/<image_name>:<tag>
docker tag <Image_Id> <New name>
docker tag 82a51cbd4876 ivanyingx/celebrities:v1

docker push ivanyingx/celebrities:v1
```





> If any user wants to run your container, they can do so directly on their local machine. In this case, you can run *ivanyingx's* image as follows:

```
docker pull ivanyingx/celebrities # to download the image
docker run ivanyingx/celebrities # to run the image
docker run ivanyingx/celebrities # runs both
```

Context

> "The build context is the set of files located at the specified PATH or URL. Those files are sent to the Docker daemon during the build so it can use them in the filesystem of

> the image." By default, the docker context is the location from which `docker build` is executed.

> When we build the image in a given directory, **everything is added recursively to the context** (so it can be copied into the image, like above)

.dockerignore

> `Dockerfile` might be surrounded by a lot of files and it takes time to copy them into `Docker` system (might even crash if there are too many files!)

> Because of that `.dockerignore` file can be specified (really similar to `.gitignore`):

Containers

> Containers are instantiations of images

> One should think of the containers as **standalone units** (like applications) **having a single responsibility**

Containers should be immutable (their internal state is always the same).

- Destroy and recreate containers quickly
- Always be in a well-defined state

```
# similar to simply running python from the command line
docker run repository/python_image:latest --help
```

> **Never try to fit everything into a single container!**

> Containers **are less of a black box**, which means attackers can analyze Docker system easier and find it's weak points

---

Commands

> Docker provides a couple of commands, which allow us to work in a similar fashion to the command line

> Docker works in a similar fashion to `git`, **it only stashes changes (additions) to the system**.

> Whenever possible **chain multiple commands using** `&&` so they are all in a single `RUN` directive

Each command creates a new layer:

- Images can be built from any layer upwards
- **Layers are cached and reused by consecutive builds**
- **Layers can be reused between different images**

FROM

> Starts **build stage** of an image

> Specifies base image (like `Ubuntu`, `node`, `conda`) which defines what one can do in the image

> `AS` defines the name for the image, like during multi-stage builds.

> It can be mixed with `ARG` (which allows us to pass this value from a command line)

```
FROM [--platform=<platform>] <image>[:<tag>] [AS <name>]

# Version is out of build stage
ARG VERSION=latest

# Here build stage starts
FROM busybox:$VERSION

# Gets version into build stage
ARG VERSION
RUN echo $VERSION > image_version
```

RUN

> Runs specified commands **during the build stage** (e.g. installing some packages)

> `shell` - if we want to run `shell` (usually `bash`) command like `apt-get install`

```
# shell
RUN <command>
# exec
RUN ["executable", "param1", "param2"]
```

> `exec` - if the base image has no shell **or** we don't want string munging

> **Main command to look out for is RUN, most of the commands (like LABEL) DO NOT create an additional layer**

ENTRYPOINT

> **Defines entrypoint (command which will be run) WHEN CONTAINER IS CREATED FROM AN IMAGE**

> Container runs as an executable

> You should always specify it (unless you want to use `shell`)

> **Either of `ENTRYPOINT` or `CMD` is needed**

```
# preferred exec
ENTRYPOINT ["executable", "param1", "param2"]
# preferred shell
ENTRYPOINT command param1 param2

FROM ubuntu
# When we run a container from the image, top -b will b
ENTRYPOINT ["top", "-b"]
```

exec d**oes not invoke shell**, hence it is not dependent on it • **Allows us to use optional** `CMD` (in a second, after command)

CMD

> **Specifies default arguments to entrypoint (if any) WHICH USER CAN OVERRIDE DURING** `docker run`

> `run` the container from the above image, command `top -b -c` will be run.

> `top -b` **will always run**

> `c` can be changed to some other flag/command via `docker run`

```
# specify executable as entrypoint, whole command can b
CMD ["executable","param1","param2"]

# default parameters to ENTRYPOINT, only those could be
CMD ["param1","param2"]

# shell form, discouraged as users is unable to overrid
CMD command param1 param2

FROM ubuntu
ENTRYPOINT ["top", "-b"]
CMD ["-c"]
```

	No ENTRYPOINT	ENTRYPOINT exec_entry p1_entry	ENTRYPOINT [“exec_entry”, “p1_entry”]
No CMD	*error, not allowed*	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry
CMD [“exec_cmd”, “p1_cmd”]	exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry exec_cmd p1_cmd
CMD [“p1_cmd”, “p2_cmd”]	p1_cmd p2_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry p1_cmd p2_cmd
CMD exec_cmd p1_cmd	/bin/sh -c exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd

COPY

> Allows users to specify which file(s) or directories should be copied into the image from the host system

> `COPY <src> <destination>`

Often idiom `COPY . .` is used, which copies files from the context location to the current working directory inside the container.

It might look like we're copying something to the same location, but that's not what's happening. We essentially have two file structures in which we are moving files between. The file system that the first argument to `COPY` refers to is the build context (wherever you run `docker build` from). The file system which the second argument to `COPY refers to is the file system within your docker container.

LABLE

`LABEL <key>=<value>`

allows us to add metadata to our image (like author, maintainer, way of contacting)

WORKDIR

`WORKDIR dir`

sets working directory to a different one

ENV

`ENV <key>=<value>`

environment variable readable throughout the concrete build stage

EXPOSE

`EXPOSE <port>`

`EXPOSE 80` would expose port `80` inside the container for others to connect (usually a person running `docker` command will specify which ports to expose and connect, hence this one isn't used very often).

Cache

Some commands invalidate cache and when this happens, every step following it will have to be re-run when you create the image

Now, no matter what happens, `python3` will be installed during each `docker build`, because Docker has no mechanism to check whether the context for COPY command changed

A**s Python installation is not dependent on the context** put `COPY` statements AFTER setting up OS dependencies

```
FROM ubuntu:18.04

RUN apt-get update
COPY . .

RUN apt-get install -y --no-install-recommends python3
RUN rm -rf /var/lib/apt/lists/*
```

Multi-Stage Builds

> **You should use multi-stage builds wherever possible!**

> First image builds the application (creating an artifact), while the second copies it and sets it up for running in a container

Pros

- Drastically reduces image size
- Simplifies maintenance

Cons

- **Mainly usable for compilable language** (sorry Python :( ) like Go, C++
- **Even better with statically linked** (e.g. everything is contained in a single executable)
- **Hard to make it work with ML/DL** as those require a lot of dependencies

```
# FIRST (BUILDER) STAGE
FROM golang:1.7.3 AS builder

# Obtain golang code
WORKDIR /go/src/github.com/alexellis/href-counter/
```

```
RUN go get -d -v golang.org/x/net/html
COPY app.go .
# Compile is as a single exectuable file called app
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .

# SECOND STAGE
# alpine is a very slim file system (few MB) great for lightweight deployment
FROM alpine:latest

# Setup only bare necessities
RUN apk --no-cache add ca-certificates
WORKDIR /root/
# Copy self-contained app into the smaller image
COPY --from=builder /go/src/github.com/alexellis/href-counter/app .
# Setup the application as container's ENTRYPOINT
ENTRYPOINT ["./app"]
```

Docker Command Line

```
# manage docker images (like building/inspecting)
docker image SUBCOMMAND
# manage docker containers (like creating from image/stopping/restarting/killing/inspecting)
docker container SUBCOMMAND
# manages volumes (persistent data storage attached/shared with containers)
docker volume SUBCOMMAND
# manage docker networks (like creating/inspecting/listing)
docker network SUBCOMMAND

# configuration
docker config SUBCOMMAND
# manage multiple containers
docker stack
# manage secrets
docker secrets
# manage docker (like space/cleaning)
docker system

# these two are the same
docker image build
docker build
```

```
# DOCKER IMAGE BUILD

# build IMAGE from Dockerfile and context
docker image build [OPTIONS] PATH | URL | -

github: docker build github.com/creack/docker-firefox
tar.gaz: docker build -f ctx/Dockerfile http://server/ctx.tar.gz

# OPTIONS
# tag the image
-t
docker build -t whenry/fedora-jboss:latest -t whenry/fedora-jboss:v2.1 .
# specify different file
-f
docker build -f dockerfiles/Dockerfile.debug -t myapp_debug .
# pass arguments to the build stage
--build-arg
docker build --build-arg HTTP_PROXY=http://10.20.30.2:1234 .
```

```
# DOCKER CONTAINER RUN
# Docker runs processes in isolated containers. A container is a process that runs on a host
docker container run [OPTIONS] IMAGE[:TAG] [COMMAND] [ARG...]
# OPTIONS = detached/foreground running/network settings/runtime constraints/command run
# COMMAND = specifies a command to pass to image entrypoint
# ARG = arguments passed to command
```

```
# RUN AN INTERACTIVE CONTAINER
docker container run
# Attach a terminal with -
-a NAME_OF_STREAM
# allocate pseudo terminal (TTY)
-t
# keep STNIN open even if not attached to CLI
-i
# /bin/bash specifies the entrypoint of TTY we have attached (or attached to)

# OPTIONS
# name the container (always do this)
--name NAME
# remove container after exit (always do this)
--rm
# run in detached mode
-d
# set memory limit
-m
docker run -it -m 300M ubuntu:14.04 /bin/bash
# pass environment variable
-e NAME=VALUE
export today=Wednesday; docker run -e
"deep=purple" -e today --rm alpine env
# override default image entrypoint
--entrypoint
docker run -it --entrypoint /bin/bash example/redis
```

```
# EXIT CODES
0 executed correctly
8 could be a 404 error
125 error within the daemon
126 contained comment cannot be invoked
127 contained comment cannot be found
```

VOLUMES

https://www.youtube.com/watch?v=1_GBQD2EZ1Q&feature=emb_imp_woyt&ab_channel=AiCore

> **Docker may create artifacts** (like metrics from training or data after preprocessing)

> **Volumes are persistent storage shared between host machine and Docker container(s)**

> data sharing between containers and hosts (example: data preprocessing container creates dataset, neural network container trains our model on it)

> we can copy to/from the containers and do the live updates of their data content

> we can set the volume to be readable only for increased security

> To get them out of a container
>
> - docker container cp
> - using volumes

```
# create volume
docker volume create docker_lesson
# mount volume to directory inside container and list its contents
docker container run --rm -v docker_lesson:/lesson busybox ls /lesson

# MOUNTING
docker run \
  --name devtest \
  --mount source=myvol2,target=/app \
  nginx:latest
```

Examples

```
docker pull postgres # A default postgres image from docker

docker run --name my_postgres_container -e POSTGRES_PASSWORD=mysecretpassword -d -p 5432:5432 --rm postgres
# docker run = command
# --name my_postgres_container = name of container
# -e = enviroment variable | POSTGRES_PASSWORD=mysecretpassword = specific to postgres
# -d = detatch = runs in background
# -p 5432:5432 = port mapping local port:container port
# --rm = removes container when it is stopped
# postgres = the name of the image the container is made from

docker ps
```

```
# Specify image:tag(or version)
# Build the image based on the default docker python image
# You can stack as many images as you want
# First will look for the file locally, if not it will pull from docker hub
FROM python:latest

# Run bash command
RUN mkdir /mydirectory

# Set current working directory
WORKDIR /mydirectory

# Copy data from docker context
# main.py = where on the local machine are you copying files from
```

```
# . = where in the container to copy them too
COPY main.py .

# Command run when the container is run
ENTRYPOINT ["python3", "main.py"]
```

```
# Build the image using everything (recursively) in the file
docker build -t custom-python-img .
# . = using everything (recursively) in the file
# -t = and give it a name

# See all the images
docker images

# Run the image
docker run custom-python-img
```