-=Udemy course:

Learn how to manage Docker hosts and Docker containers through project-based training.

Instructor:jason

Types of docker editions:

1. Community edition or CE (Free)
2. Enterprise Edition or EE(not for free and has some premium features)
3. docker Enterprise

docker Update channels:

1. Stable- gives you latest releases for general availability
2. Test- gives pre-releases that are ready for general availability **after testing**
3. Nightly- gives you latest builds of **work in progress(WIP)** for next release

Stable version is generally preferred as it is less prone to issues

To install cleaner installation of community edition of dockers remove older versions of dockers from your system, in case you have some data that needs to be retained, **take backup(and store it somewhere else) of var/lib/docker where docker data is stored**

* To install/ uninstall you need to have root user access or admin credentials
* docker can be installed on any name ex: docker or docker-engine or docker.io or containerd or runc
* to remove docker we have to include all the names that we used in above point, and use admin or root mode

apt remove docker docker-container docker.io containerd runc

**Add the Official docker Repository**

Become root user first or use sudo -i command

Make sure that all your packages are up-to-date by executing these commands:

apt update

apt upgrade -y

Next, ensure the prerequisites are installed by running the following commands:

apt install -y apt-transport-https ca-certificates curl

apt install -y gnupg-agent software-properties-common

NOTE: Many times these packages are already installed. If that's the case, apt will simply report

that they packages are already installed.

Install the GPG key provided by docker:

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | apt-key add -

Verify that the correct key with this command:

apt-key fingerprint OEBFCD88

Verify that you’ve installed the key with the fingerprint of:

9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88

Do this by searching for the last 8 characters of the fingerprint.

apt-key fingerprint 0EBFCD88

If a key is returned from that command, then the correct key was installed and you can proceed.

**For ubuntu the docker software repositories correspond to the code name of the distribution**, get that by running the command:

lsb\_release -cs (c stands for code name and s stands for short)

(it can be bionic or focal or some other thing)

Now, add the repository with this command:

add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb\_release -cs) stable"

**Install docker CE**

Update your local packages index:

apt update

NOTE: If you see errors such as "The following signatures couldn't be verified because the public

key is not available," then re-run the curl command from above to install the GPG key properly.

Install the docker Engine and its associated packages:

apt install -y docker-ce docker-ce-cli containerd.io

**Test the Installation**

Check that the docker client is available and working by running the following command:

docker version

* **Docker Daemon**

The Docker Daemon runs on your host operating system. This is typically your main computer or a server on the cloud. It currently only runs on Linux but there are ways to run Docker on MacOS and Windows too.

You can think of this as a service that runs in the background. It is the brains of the operation when it comes to managing Docker containers.

### Docker CLI

The [Docker CLI](https://nickjanetakis.com/blog/understanding-how-the-docker-daemon-and-docker-cli-work-together) is one way to interact with the Docker Daemon. The Docker Daemon exposes an API and the Docker CLI is one tool you can use to consume that API.

It’s imperative that you understand how to use the Docker CLI, but in your day to day activities you’ll likely use another tool called Docker Compose, but that requires knowing how the Docker CLI works.

**Use case:**  
When you want to CRUD (create, read, update, destroy) various components of Docker, such as your Docker images, containers, networks and volumes.

Docker sub command groups:

Docker image- to manage container images

Docker network- to manage docker networks

Docker container – to manage containers

Docker volume – to manage docker volumes

Docker swarm – manage docker swarms

Docker service – to manage docker swarm service

They may not seem much self explanatory but, by seeing few commands you might be able to grasp what the sub command group related commands can give, docker image ls -> lists all images that are there, docker rmi – to remove an image, docker container ls -> lists all containers, docker container rm -> to remove container

Use case:

Provides core Docker functionality. It must be running on every host you plan to run Docker on.

**what is container and what does it do**?

* Containers are designed to be short lived and temporary
* A container is a standard unit of software that can run a particular application and its associated Processes
* A container **contains application process, file system and ip address (also called as network stack- to communicate eith other containers or computers)**
* Container are mainly used to enclose application in controlled and configured environment
* When a container is created, directories and files are populated into its file system from docker image from which the container is created, this is the way in which all the software binaries that are to be installed are stored in container
* A container is portable and contains software dependencies for a particular application
* docker did not invent containers, there were linux containers around for a while, docker just made containers to easy to work with
* Container can be used for a task or even for a single process (or a small group of processes) to provide a service, so It is common to run several containers at once on a single host machine, whereas vm’s can be used to run multiple processes and tasks, this makes it obvious that containers take less boot time compared to virtual machines
* One great benefit of running docker on linux containers is that, it is not disruptive to a system in the way some packages are disruptive to the system after they are installed. i.e, they are self contained and do not alter the host system after they are installed.
* docker is light weight
* application running in a container can not access libraries that are present in other container or host system.
* Applications running in different containers can have same port number as they are configurable because of the different ip address they bare because of the containers they are in
* Container can be
* A container runtime unit-providing application isolation
* Engine-os kernel: helps in isolation of container
* Management tools: cli, gui, api server
* Above engine and management tools will help in **performing all type of operations one can do with container** such as create, start, stop, delete containers, list containers, manage container volumes and networks, read logs, performance monitoring, container image management etc.
* Example of container run time is **docker engine, docker desktop and docker machine.** Docker help us to do all the above mentioned operations,there are other container runtimes available,
* Fire cracker,containerd,rkt,podman
* Containers provide consistent and reproducible environments, if it works in development cycle, it will work in production cycle as well
* Unlike virtual machines which have complete os and setup installed , a container is few megabytes in size and is easily transferred between servers
* Using containers reduce the number of servers required, cutting down the host machines that are needed
* Orchestrators can be used to organize and run multiple containers, making sure they are performing different functions efficiently. Ex: docker swarm,Kubernetes(used to handle multiple workloads) or red hat openshift, amazon elastic container service, google kubernetes engine, azure Kubernetes engine, apache mesos and so on...
* Container images that are developed in development environment are deployed on top of above orchestrators in production environment
* **Container image:**
* Container images are large blobs, which are required by container runtime to start and run a container
* It is easy to create, store and **run container images** with docker
* It is a basic deployable run time unit
* They are used to deploy our application or services in different environment easily
* Container image=file system+metadata
* For example a container image related to a python project will have python interpreter+pip+source code (of project)+libraries/modules+os utilities and files
* DOCKER used aufs for container file system layers, now it is uing overlayfs container file system layers
* A compressed snapshot of an file system along with its meta data(a container image has everything to run the application)
* Naming convention- <registry>/<repository>:<tag>
* <repository>:<tag> form the actual image name
* <registry> shows from which registry is the image repository originates
* A **container image** is required to start a container and the images can be stored in an online library called image registry (ex: docker hub), it is preferred to have own private registry than using existing one.
* Container image can run the same on windows, macOS or linux.(no more **it worked on my computer 😊**)
* Python code packaged in to a container image on your computer runs the same on any computer or even on cloud environment
* Image lifecycle: source code+datafiles+config files-> include them in docker file->docker build command to build image -> push to registry
* Container life cycle: a image from localhost or from registry is pulled and then container image is created, it can run, from run it can move bidirectionally either to paused or stopped, from stopped state it can be killed
* Full container life cycle = image life cycle + container life cycle
* Build once run anywhere
* by using exec command we can make changes from inside the container, this is also called as **entering a container which is already running, we can use this exec command to running or paused containers**
* **syntax:** docker container exec <container> <command> or docker exec <container> <command>
* **options:**
* -it for interactive commands
* -w /directory to set working directory
* -e var=value to set environment variable for the command
* docker run -dit --name accessing\_Container\_from\_inside
* now we created a container, next we will use exec command
* docker exec -it accessing\_Container\_from\_inside /bin/bash
* if you notice at first command we used -dit and in second command we used -it, now we can use the container interactively
* now you can exit by typing exit
* you have exited, but you can now reconnect(**or enter in to the container**) again using the same command above
* docker exec -it accessing\_Container\_from\_inside sh
* we only used sh, instead of /bin/bash, but it will still work
* docker exec -it d96 bash
* docker exec -it d96 /bin/bash
* above both are right as well
* **using -it is very important, without them, the shell will execute and immediately exit**
* Now we will try to create a detached container(using -d) and then creating an empty text file inside it using exec command.
* root@TIGER02143:~# docker run -dit --name execution httpd
* 048d90bdbb325df6457c9abb43cb4ef8bcb1e7801eb22ae898743d0620296f14
* root@TIGER02143:~# docker exec -d execution touch /root/hello
* now to list contents in root, you can do it in 2 ways
* by going inside container and then executing ls
* docker exec -it execution sh
* ls /root
* directly by one command
* docker **exec** -it execution ls /root
* to stop a docker container or to start a stopped container use
* docker stop <container name or container hash > or docker container stop <container name>
* to create a container **without starting it**, use docker create <image:tag> or docker container create <image:tag>
* you can use the same options as you use docker run command
* this create command just downloads image from registry if not available in local image cache
* container files system is created, as well as its control records (similar to docker run command)
* to start a created container use **docker start** command
* when you are trying to assign a port to container create command, make sure you are not assigning existing port to it
* to start a created container use **docker container start <container> or docker start <container>**
* this command starts container and executes **startup command** as well
* we can start a container as many times as we want, but
* the startup command executes each time the container is started
* how ever the files that are created or changed in previous start sessions will be available in file system
* starts created or stopped container
* -a option to attach to container stdout/stderr
* -I option to attach to container stdin
* Using -ai will start container in foreground interactive mode
* If you are having port assigned to container, and it is in stopped state, if you try to start it using tcontainer start command, then you must check wether the port assigned to this command is not assigned to any other container after the port(which you are trying to restart) is stopped. using **docker ps -a** command
* Docker inspect <container name> or docker container inspect <container name>
* Gives detailed information about a container in json array format
* It has various information of container related to status, network,config and many more
* Similar to inspect of container we can have inspect for a image and look at its metadata contents
* docker image inspect <image> or docker inspect <image>
* this gives metadata information about image
* docker image history <image> or docker history <image>
* lists the build instructions on how the image has been built
* works on images present in local image cache
* if not present in cache needs to be pulled and then run history command
* Docker pull <image> or docker image pull <image>
* Pulls image from registry, if the same image with specified tag is not there in local image cache
* You can speed up container initialization by pulling image before hand
* If you don’t specify the tag, latest tag is assumed for the image and is pulled
* -a or --all is used to pull all the images with different tags , but we need to consider disk space and time
* We can pull from any image registry ex:google container registry,azure container registry
* Rename image:
* Docker image tag <image> <new name> or docker tag <image> <new image>
* Adds <new name> to existing image
* One image can have multiple names
* To remove image name use docker rmi
* Environment variables in container:
* We can initialize a set of values to a variables during a container is created using some docker command, new variables can be assigned or overridden after the container is created as well
* docker run -e VAR= val [options] img [cmd]
* docker run -e VAR [options] img [cmd] (this type of declaration will let check wether the mentioned VAR is already present or not, if present that value is passed to the container for env variable, else it will just be ignored)
* docker run --env-file file img cmd
* alternately you can create a environment variable out side and assign some value and later on when you create a container you can give the name of variable and assignment is done
* $Env:MY\_VAR=”test\_value”
* docker run --rm -e MY\_VAR python:3 env (env here prints environment variables as ouput after the command is run)
* we can assign or reassign using -e option, using --env-file to read environment variables from file in docker host
* Removing container
* Commands:
* docker container rm [-f] <container name or unique id>
* docker rm [-f] <container name or unique id>
* docker container prune (removes all stopped container)
* container file system is permanently deleted
* rm command is used to remove stopped containers
* -f option is used to kill running containers
* Prune is used to kill all stopped containers
* Portainer: (https://www.portainer.io/)
* An gui for docker where we can almost manage all the activities we perform using CLI
* Will be useful to get familiarize with the concepts of docker and its uses
* Prima facie user friendly
* We can set portainer by running the below commands
* First, create the volume that Portainer Server will use to store its database:
* docker volume create portainer\_data
* Then, download and install the Portainer Server container:
* docker run -d -p 8000:8000 -p 9443:9443 --name portainer --restart=always -v /var/run/docker.sock:/var/run/docker.sock -v portainer\_data:/data portainer/portainer-ce:2.9.3
* open <https://localhost:9443/> you can login by giving password and do the needed things there
* steps in manual building of image(python):
* create a temporary container with python base image with -d detach mode and later invoke with -it option
* docker run -d –name <container name> <base image>
* docker run -d --name temp jupyter/tensorflow
* docker exec -it <container name> <command>
* ex: docker exec -it temp pip install torchvision torch
* final step is to commit or freeze container using
* docker commit <container name> <name that you want to add to final image>
* docker commit temp myjupyter:torch
* docker commit -c “WORKDIR /app” -c “ENV FLASK\_ENV=development” -c “CMD python script.py”
* here -c is used to make changes in metadata settings i.e, working directory to /app, second -c is used to set environment variable, and cmd is the startup command (if entry point is empty, cmd alone executes, else it will be executed along entry point command)
* now you can use the name that you used to add it as final image and then create a new container using the above image as base image
* docker run -it –rm --p 8888:8888 -v ${ pwd }:<path> myjupytertorch (image used above)
* this will launch a jupyter session on port 8888
* execute
* **Container registry:**
* Dockerhub is main example of **container registry**, used to **store,share and manage** container images
* Docker hub is the largest resigtry in terms of repository count
* Docker hub is home to many official images (which are published by docker, ex: python, java and so on..)
* In python alone we have many images- python, pypy,tensorflow(by tensorflow community),pytorch(by pytorch community), centos/python-36-centos7 (if you are well versed with yum utilities)
* Ways to build images:
* Manual image creation through docker commit
* Automatic image creation through docker build
* Alternative tools:
* Buildah,orca,kaniko
* CI/CD tools:
* Github,gitlab (docker hub can integrate with github through web hooks so that each time source code is pushed to repository a new image trigger is created on it)
* You can create a **docker image** in 2 ways

1. Manually by having base image, source code, data, and config files
   1. Pull the image
2. By using special script(mentioning custom requirements) Dockerfile with dockerbuild command

* A dockerfile is a script with instructions on telling docker build command on how to assemble file system with metadata
* Syntax1:

#comments

From base-image #basically the starting line of docker file(anything else will raise an error)

INSTRUCTION arguments

INSTRUCTION arguments \ (the backward slash denotes next line is continuation of this line, considered good programming language)

More arguments

INSTRUCTION arguments

Syntax2:

#comments

From base-image #basically the starting line of docker file(anything else will raise an error)

INSTRUCTION arguments

INSTRUCTION arguments More arguments

INSTRUCTION arguments

* Commands used in creation of docker file:

FROM <base image>- Sets the base image. (checks whether the image is present local image cache, if not tries to pull from registry), all the properties(file system,working directory, mount points, metadata definition, binaries, system utilities) of base image are copied in to the current image which uses this dockerfile, selecting a base image is very important because if we can select a base image which already have the majority of the python requiremts already covered in the base image, it will save a lot of time and effort

Ex1: from python (latest version is considered as tag is not explicitly specified)

Ex2: from python:3.8

Ex3: from localhost:5001/mypython:3-slim

Docker build command will be able to successfully build images as long as the version is supported and is available from image registry(ex3 approach will work id the local registry is configured in the same way in other machines where you want to use this docker file, because if there is any misconfiguration or different port number being used, then it might throw error), if image is not available we might have to use some ci/cd method or some other approach to update to the image version available or it is better to not to mention the tag unless needed.

WORKDIR <directory path>- declares working directory for rest of docker instructions(if mentioned explicitly again, then the new assignment is used as working directory for further instructions) and startup command, if not changed explicitly workdir of base image is used as working directory, workdir is nothing but when the container starts running, then the cwd of container is set to the path that is assigned earlier to cwd

Ex: if you give WORKDIR /my-work-dir, when the container is running, it will look something like below root@f5e....

RUN <software and libraries installation>

Or

RUN <initialization commands> - Executes a command to help build your image.

RUN <COMMAND> - here the command is executed as shell line command(a normal command you give in a linux terminal like ls or pip install or python script.py)

* Example commands
* RUN pip install -r requirements.txt
* RUN pip install numpy=3.4.4
* RUN conda install -y pandas
* RUN python script.py <param1> <param2>..
* RUN user add chris && \ (here we had used backward slash to split in to 2 lines)

Chown -R kris /app > /tmp/logfile (here we redirected data using   
“>” operator, also we used “&&” for command pipelining,while pipelining if any command fails it will stop execution and returns non zero code as status)

EXPOSE <container port>- Opens up networking ports.

VOLUME <mount point>- Sets a disk share. (din dig deeper)

Commands:

* Docker volume ls
* Docker volume create <name>
* Docker volume rm <name or ID>
* No command to rename volume

COPY <folder (which resides at same level as dockerfile does)> <destination path> - Copies files into your image from the local disk recursively.

* COPY source destination
* COPY source1 source2 …. Sourcen destination/ (here it is ending with / because we are copying multiple files and destination should be a directory for sure)
* **If destination is directory it must end with /**
* Source and destination should follow linux path format(forward slash system)
* Source and destination can be absolute or relative paths
* **Build context** is the current working folder and root folder for source
* If source is a folder, its contents are copied recursively to destination directory
* Destination folder is created if it does not exist
* If you try to copy from src directory to a target directory, then all the contents of source directory are copied to target directory
* You can also change modes of permission(din dig deeper)
* Copy command also allows wild card entries
* Copy \*.py <target>

LABEL - Adds metadata to an image in a key-value pair format.

ENV - Sets an environment variable (defines variables to be included in startup command)

VOLUME <path>: declares mount point

EXPOSE < port >: exposes network port that communicates with running container

ENTRYPOINT [“command”,“option1”,”option2”….]- Determines which executable runs when the container starts. (Use CMD to to pass options to the executable.)

CMD [“option3”,”arg”]- Sets the command to be executed when running the image. Can also be used to set the default arguments to the ENTRYPOINT instruction.

Use entry point to define a default executable and CMD to declare default parameters for this command, CMD can be easily over riden by container creation or run command by passing required string after image name in the command

If ENTRYPOINT [“**command**”,“option1”,”option2”….], CMD [“option3”,”arg”]

START-UP COMMAND= **command** option1 option2 option3 arg

**Command**  used her must be an executable i.e, a binary or a shell script with executable flag set

Ex: CMD [“python”, “script.py”] (will be executed as python script.py)

Let say that you have a startup command for container xyz

If you use command **docker run xyz** then the startup command it has will be run automatically

**All the above commands are used in assigning meta data to docker file**

**Ex: cat Dockerfile (d should be capital)**

**Syntax:**

FROM image\_name:tag

LABEL maintainer =<’name of maintainer’> (key value pair type assignmenrt)

ENTRYPOINT [“<entry point information>”] (json array type assignment)

CMD [“[<command information>](http://www.docker.com)”]

**example:**

FROM Debian:Latest

LABEL maintainer =<sai\_vinil> (key value pair type assignmenrt)

ENTRYPOINT [“/bin/ping”] (json array type assignment)

CMD [“[www.docker.com](http://www.docker.com)”]

Using the above Dockerfile you can create an image from it, by using the command’(docker build command is used to enable automated container image creation, docker build takes docker file from host and builds a new image), docker file , docker build command first builds a container and then executes the commands in the docker file (build context- a directory in development host where you will store dockerfile other files which you want to move to image file system( using copy command))

docker build -t <docker\_username>/<page(also called repository)>:tag <path were we can find the dockerfile> [-f <Dockerfile >] <build\_context> <path of the build context(generally passed “.” Which represents cwd)>

-t ->sets name and tag of final image

Build\_Context -> the folder where all the files and folders(sub folders as well) required for **docker build command** reside, **docker build** command sends contents of build context to docker host each and every time build command is run, so to avoid large files (or even unwanted small files or **folders**) from being sent each and everytime, you can mention them in **.dockerignore (similar to .gitignore)** to prevent from being sent to docker host

-f (this is used when we don’t have the docker file name as Dockerfile, instead we have multiple docker files, such as simple.Dockerfile, example1.Dockerfile in such a cases we will try to use -f option to explicitly specify which image is to be used) provides non standard docker file name, must be in build context folder, default name is <build\_context>/Dockerfile

ex:

docker build -t vinilta/debian:latest .

the above “.” Specifies the Dockerfile location, you don’t need to mention the name of the docker file if you name the file Dockerfile, if you give another name, you have to specify the name of the file ex: dock1.dockerfile (not sure of the convention used here)

when you execute **docker build** command using dockerfile, each instruction in dockerfile is executed as a step, i.e, 6 instructions in dockerfile means 6 steps are executed while running docker build command

When a container is started based on this docker image, the first command that will be running is **/bin/ping** [**www.docker.com**](http://www.docker.com) **(information passed for cmd is nothing but the parameter and it is similar to python)**

* After creation of docker file you need to create a docker image:
* docker build -t <docker userid or namespace>/repository:<optional-tag(if not given it will by default use latest)> <path of directory where docker file is there(this file is used to build docker image)>
* ex: docker build -t <docker id>/dockerping:latest .
* here the . specifies the docker file is in current directory
* -t will allow us to specify tag name or image name

Build and Push an Image

***Goal:***

The goal of this exercise is to create an image based off of a Dockerfile you create. You will also

host this image on Docker Hub.

***Instructions:***

**Create a Docker Hub Account**

Create an account on Docker Hub. This account will allow you to host your own images.

Visit https://hub.docker.com/signup in your favorite web browser.

Follow the instructions on the screen by choosing a username (Docker ID) and a password. Be sure

to accept any items such as the privacy policy and terms of service.

**Create a Dockerfile**

Using Debian as the base image, install the nano editor package. Also, have bash start by default

when the container based off of your image is executed.

First, create a temporary directory to work in.

mkdir tempbuild

Change into the directory you created.

cd tempbuild

Create a Dockerfile by opening it with a text editor such as nano . (NOTE: Use your favorite text

editor for the operating system you are working on. For example, nano is not installed by default on

Windows and some Linux distributions. For example, you can use notepad for Windows and

TexEdit for Mac.)

nano Dockerfile

http://www.LinuxTrainingAcademy.com

Enter the following contents into the file. **NOTE** : It is very important that the contents are exact.

Capitalization, punctuation, and spacing matter!

FROM debian:stable

LABEL maintainer="ENTER\_YOUR\_NAME\_HERE"

RUN apt update && apt install -y nano && apt clean

CMD [ "/bin/bash" ]

Save the Dockerfile. If you are using nano, type Ctrl-o. Next hit "Enter" to save the file. Finally, type

Ctrl-x to exit the nano editor.

**Build the Image from the Dockerfile**

Build the image using your Docker Hub ID as the name space, "nanotest" as the repository, and

"latest" as the tag.

For example, if your Docker ID is "robertsmith", then use "robertsmith/nanotest:latest".

docker build -t YOUR\_DOCKER\_ID/nanotest:latest .

NOTE: You could also use "YOUR\_DOCKER\_ID/nanotest". Remember that the "latest" tag is used if

no tag is specified.

After the build completes, check that the image is available on your local Docker host:

docker images

You should see the nanotest image is present.

**Start a Container Using the Image**

Test that your image starts bash and contains the nano package as expected.

docker run --name nanotest -it YOUR\_DOCKER\_ID/nanotest

You should be presented with a bash prompt. Now, see if nano is available.

which nano

If nano is available, the above command will report the path to the nano executable: "/bin/nano"

http://www.LinuxTrainingAcademy.com

Exit out of the container.

exit

**Push the Image to Docker Hub**

Login to the Docker Hub registry using your Docker Hub ID.

docker login

Provide your ID and password when prompted.

Push your image to Docker Hub by using docker push command.

docker push YOUR\_DOCKER\_ID/nanotest:latest

* Base image- a tiny but functional operating system, docker uses layered file system , so that you can add to base image

From Debian: stable (the debain stable image is being used)

* Why containerizing an app is important
* A small difference in the version of an external library can change the functionality of your application, thus, causing it to behave differently. Therefore, containerizing an app allows it to execute in the same way regardless of the workspace or computer that it is deployed on.
* The beauty of Docker is that if you containerize your application and transfer the image to your colleague’s computer, you can be sure that the application will have the same performance on both devices. This is because the container includes all the application’s dependencies.
* Container and virtual machine difference:
* Container does not have OSKernel(all containers have same kernel), whereas VM does have(each vm has its own kernel)
* VM more occupies cpu, disc space, and more memory than container because of its own kernel
* Container can be archived and packed in current host machine and can be unpacked and executed on another machine very easily
* Parts of container:
* A container has 3 parts:
* Application process, file system and ip addresses.
* a
* How do containers communicate:
* They communicate using network protocol(by default and can be changed explicitly), which is based on ubiquitous internet protocol(IP)
* Container and host machine are connected using **virtual bridge/switch(which is created by docker engine and is connected to network card, which allows communication with host machine and outside the machine as well),** virtual bridge/switch identifies containers based on IP addresses.
* Containers can communicate with normal applications running on host machine. i.e, container ip address-> virtual bridge/switch -> host ip address -> application installed on host

(computer on which docker is installed), they can also communicate with other containers, provided that they know the ip address of the container.

* A container can communicate with another container(through ip), though it can be controlled by network segmentation and access control methods
* You can also create a virtual network for containers, so the containers created in the network can communicate through hostnames, instead of ip addresses
* By default configurations available, a container can communicate with a network server(a good example for this is when you do pip install command it can download the required package provided that computer has network connection)

Container ip address -> virtual bridge/switch -> network card (inside host machine) -> network server (outside the host)

* A jupter notebook(a network server application running in a container with web browser running on a host machine) launched from a container(Jupiter network server) is an example of web server application
* Here a jupyter running in a container can launch jupyter notebooks on host web browser, this is because , docker facilitates **port mapping mechanism** , the docker takes host machine port and maps it with container port(**here host machine does not have any idea of container port number detail**), so the data transferred to host machine port can be automatically transferred to container port
* **How containers store data?**
* This is important because,Containers are isolated, they don’t have access to host storage, software binaries, utilities, so containers need dedicated file system to store at least needed file systems
* When container is created from an image, files and directories are populated from the image
* A container can store its data in file system
* Container can be stopped and killed, but the data inside is stored in host machine file system with the help of bind mounted folder, this way data stored in mount directory survives removal of container
* Data is moved in to container file system through a process called **bind mounting**, where we can transfer sub directories, folders inside a directory (in host file system)to a directory (inside the container file system)
* Docker and other container run systems support volume mount mechanism, in which volume of independent database with its own file system can be mounted to a mount point in container, the volume persists all the data that is stored even after the container is removed, the volume can be mounted to another container making its data available to the newly attached container
* Data cannot be transferred bi directionally between volume and host directly, to transfer you need to create a container with both volume mounted and bind mounted , so that using linux shell you can transfer data between them
* Microservices:
* An architecture that considers application **as a collection of loosely coupled services**
* easy to test and fix
* new microservice can be deployed without deplyoying the whole application
* docker containers are ideal for micro services
* **docker security:**
* docker isolates one container from another
* docker has no built in data protection tools
* Multiple containers on a **host machine** will share the kernel of the host machine (a potential threat where if one container compromised, it can potential hack risks to other containers , in the worst case whole host itself)
* Uses isolation techniques such as cgroups(control groups) and namespaces
* Basic docker commands(same on linux, windows or mac):
* docker run -dit Debian (it will create a container, here debain it is the type of image you created)
* docker run=docker create (creates a container which has only file system)+docker start (starts contianer by assigning ip address)
* **docker run or docker startup command will create container from the startup command(which is a combination of entrypoint and cmd key values in config of metadata json array) that is present in the image metadata based on which the image is going to run**
* use command: docker run [options] image new\_cmd or docker run [options] –entrypoint new\_entry image new\_cmd to over ride the above command and give new custom commands and entrypoint values
* **docker run** command starts a container
* -d stands for detached, it allows for a **container**  to run in the background, when a container is run in the background, the docker will print container id(in the terminal)
* -I stands for interactive, keeps stdin and stdout for a container open even if the you are not attached to the container, it allows you to send commands in to the container
* -t stands for terminal, in order to have an interaction with container you need terminal 😊
* -it is used to create an interactive container, we can type commands and see the output in terminal, we can press ctrl+c to stop interactive mode
* If you want to start the container again, you can simply give
* docker start -ia <container name>
* If you simply run “docker run Debian” it will just open a container and close it. So it don’t even return the “container id”
* it isn't necessary to use -it with a container running a service. If it's a full OS that you would work with by connecting to its terminal and using the shell, then "it" would be necessary for the container to start - the OS would need an interactive terminal to connect to. For an nginx container, on the other hand, you don't need to access a bash shell inside the container - instead you would interact with the web server by modifying files the container uses.
* Check the different type of declarations above
* root@TIGER02143:~# docker run -dit debian
* 20b1acccd6115ab34f132ad73af5c8b4ef845b5b22814f339b1df06c12a79e78
* root@TIGER02143:~# docker run -d -i -t debian
* 1de18d55fa3aabbab3e82f73992f255b9fb6537ac7066bec6a1ecaaaaa37d28d
* root@TIGER02143:~# docker run -d -it debian
* 24f660d216233b15a9ecc0ce228e16467da66df3518381f8b0d71b9150d46990
* root@TIGER02143:~# docker run --detach -it debian
* 3d6120665e365d79729ee52886a4b4cc578e154bf83ede6680e79e548e3c4551
* To check the information regarding the container use: “docker ps” command, which will list all running containers or you can use docker container ls, if you pass additional -a command it will return all paused and stopped containers as well
* It will give CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
* Container id- first 12 digits of container unique id
* Image- name of the image used to create the container
* Command- displays container startup command
* Created, status fields are pretty explanatory, if the status field is bearing “created” value for a container, then that means the container is created but not started yet
* Ports- shows host port->container port information, and is only available to active containers
* Names- this section will have the name of the container, name can be customly declared by us, if not docker assigns some random name, can be renamed at any point, automatic names will have **adjective\_personname** format
* To stop a container
* Stop <container id or name of the container>
* Command:docker images
* displays what images has been downloaded by docker
* docker images --no-trunc - will display hash information without truncating it
* docker inspect <full docker id or 3-4 starting digits of it>
* gives a detailed information about the container whose id is passed
* **Managing Container Images :**
* Takes less space
* If you want to create a new image with tag from an existing image which already has a tag
* docker tag < old\_image>:<tag\_of\_old\_image> <target\_image\_name> : <tag\_you\_want\_to\_give\_to\_this\_target\_image>
* hash will be the same for both old and target images
* docker history <image name>: this will give information on various layers that are involved in getting the current image
* each and every time you use a run command, a new layer is created, to avoid new layer you can join 2 or more commands In one run using && to join them
* in general it takes less time to building an image is quicker if you have less layers
* Debian and ubuntu distributions use apt commands whereas redhat linux and centos use yum commands
* docker rmi <image>.<tag>
* here rm stands for remove and I stands for image. The above command untags(indirectly removing it) the specified image
* if it has layers associated with it, they will be deleted.
* You can use -f option to above command if you want to use it by force
* The image layers that are not being used by **tagged images** are called **dangling images**
* docker image prune (use it with caution)
* this will remove all dangling images and frees up space
* to remove both unused and dangling images give
* docker system prune -a (use it with caution)
* docker system df:
* use to check the disc space used, assume df for disk free or disk fragmentation
* You can determine the name of the image by searching for on the command line with "docker search TERM" or by searching on <https://hub.docker.com>.
* If that tag becomes outdated and doesn’t work for you then visit this page on docker Hub to get a working tag:
* https://hub.docker.com/\_/httpd?tab=tags
* if you don’t specify tag, docker performs operation with ‘latest tag’
* when you use docker run -dit debian , you are not giving the container that is being created a name, all it will have is a hash to identify it and a random name that docker assigns to it, but you can also give it a name using the --name option
* docker run -dit --name= ‘first\_name’ Debian
* docker run -dit --name redis\_container redis
* above both are valid declarations
* docker ps -a:
* lists all containers which are running currently
* docker ps -l:
* lists latest container regardless of its (running)state
* all the docker container related commands start with **docker container <options>**
* ex: docker container ls – lists all containers
* multiple docker containers can not have same name
* to remove a docker container use
* docker rm
* you cant remove contain directly, stop it first and then remove it
* almost any command that takes hash(container id) as input will take container name(custome generated one or docker generated one) as an alternative to perform operations on it.
* If docker engine rebooted or the if container crashed because of some reason, but you want it to restart automatically , then use:
* docker run -dit --restart=always --name=fourth\_container debians

You can check information related to it in ‘docker inspect‘ command

docker inspect forth\_container | grep -A3 ‘RestartPolicy’

The above command -A option followed by number informs it will return the matching line and the consecutive 2 lines

* You are in a situation where you are unable to stop a container, then you have to kill it, but killing a container most of the times will lead to not having the possibility of restarting it.
* docker kill <container\_name>
* docker system prune
* - all stopped containers
* - all networks not used by at least one container
* - all dangling images
* - all dangling build cache
* Suppose you want to run a container and delete it automatically after It is stopped, for that you have to use --rm option, if the container is not there already, docker will try to download it, here hello-world is an image (like Debian is an image, hello-world is an image as well ) The hello-world image simply outputs some information and then stops(it runs and automatically **stops or dies**after it runs). Unlike some other containers, it doesn't provide an ongoing service, as you are not giving -d option alongside run, it will run in foreground
* Docket run --rm hello-world
* If you want to run the image in background(by using -d option) and want to see output, then you have to check logs, docker engine captures stdoutput and error to logs
* docker logs <container name or hash> or docker container logs <containerx>
* To check the output of an image that you are using with the container from the inside, then you can use the above command
* Useful Options for logs are -f (print logs terminal in realtime) and -t(execution timestamp for each command in container) --since <time> ex: --since 1h3m5s will generate logs from that time to the current time check the help options if needed
* If we start a stopped container, new logs are appended at the end of file
* Removing a container removes the logs as well
* Docker attach:
* Syntax: docker container attach <container> or docker attach <container>
* Useful when we want to interact with container and check its output
* Ctrl+c stops container, ctrl+p then ctrl+q disconnects the container
* Useful when we are having a python container and if we want to check output of it as well
* Docker machine -> this command is used to manage virtual machines with docker runtime
* docker run --name our\_nginx -d -p 8080:80 nginx
* -p (also used as --publish)used for port configuration, you can use multiple ports for same container, if you fail to assign port at the starting, you can not do it later, only way is to remove the container and start it all over again, docker port <container name>- used to displays port mappings , (each container has its own network information such as ip address and so on.. the requests that come to this server will be at port 8080 and be redirected to 80, can be accessed by url <http://localhost:8080> or in other words **localhost sends request to 8080 which forwards request to container port 80, the reply from container port is sent back to host port 8080, which is sent to localhost**) this will get an container with nginx image 8080:80(container port is 80, host port(development port) is 8080 , it maps container port 80 to development host port 8080) open up tcp port 8080 (outside world of container) on docker machine and redirect traffic to and from 80 (inside world of container)
* docker ps will have the following ouput to the above container
* root@TIGER02143:~# docker ps
* CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
* 77fd5a2c6144 nginx "/docker-entrypoint.…" 22 seconds ago Up 20 seconds 0.0.0.0:8080->80/tcp our\_nginx
* Curl command is one of the way to talk to the container and nginx takes responsibility to respond to our query for the above container
* curl <http://localhost:8080>
* it will spit out the html format
* **bind mounting:**
* **docker run -it -v< local machine directory path(or host path)>:<container target directory path (or mount point to store files)> python:3 bash**
* **ex: docker run -it -v {$pwd}:/app python:3 bash**
* host path should be an absolute path
* if there are any error in declaration of host path or container path, docker attempts to create it and because of this we might end up with an empty folder instead of the folder and files we wanted, proper check is needed in such case
* bind mount is the process of transferring files from host to container
* can be declared only during **docker run** command, if want to declare later, we cannot, only way to do that is by removing the container and creating again
* alternative option is to use --**mount (at least using 3 parameters- type,source,destination is a good practice)**
* **syntax: docker run -it --mount type = <mount type> ,source = <host path>, destination = <mount point or container path to store files> image:tag <command>**
* docker run -p 8080:80 --name another\_nginx -v ${pwd}/webpages:/usr/share/nginx/html:ro -d nginx
* let us assume we had created a web pages folder in current directory and inside it, we had crewated a simple html file.
* in abv cmnd
* ${pwd} is replaced current working directory
* pwd/webpages -- will simply point the html file
* -v -- creates volume for sharing files (in the above example a folder called usr/share/nginx is created which will have the copy of folders from the folder webpages, which is in the current directory, and container can read and write to files inside it, this is called **bind mounting**)
* /usr/share/nginx/html/: (docker container) where you want the data to be accessed form inside the container
* -v ${pwd}/webpages:/usr/share/nginx/html:ro here the folder webpages in cwd will be mapped to /usr/share/nginx/html inside the container and is given read only access
* ro are the options for above path (/usr/share/nginx/html/) which stands for read only, i.e, files can only be changed from docker host machine, but container cant alter but only read data contained in the volume
* above command maps from host to docker container

**communicating from inside the container, this is also called as entering container at runtime:**

* root@TIGER02143:~# docker run -it --name apache httpd /bin/bash
* -it is short form for –interactive –tty that means it is going to open an interactive terminal through which we can interact with container
* this command is used to create a docker container which runs in foreground and we can communicate with terminal
* /bin/bash is the command that is executed once container is created(I assume) and bash will create a linux shell called bash
* root@ffd4e8c5f6b1:/usr/local/apache2#-> its structure is

user@host name of container:cwd, basically this is inside the container, and all the commands (such as ls,pwd and so on,,,)can be used to do make changes you want to do with container

* to exit out of this container simply use the below command
* exit
* when creating a container using **docker run** command if you don’t pass -d option, the container is stopped and you cant see it when you execute **docker ps**
* use -d will look something like this
* docker run -dit --name apache httpd /bin/bash
* but when you use -d you can not interact with container like you did in above case
* we want to communicate from inside the container, but you cannot if you don’t use -d and if you use -d, it will start running from background, so we need to use another way from which we can communicate with container
* as the docker containers are slimmed versions of os, they might not support all the commands that an OS does
* ex: man, vim, vi, nano are not available in container
* but you can install them based on the distribution (if you are using Debian, you use apt commands, if you are using red hat linux, you will use yum commands)
* ex: apt install -y vim
* installing a package in a container does not reflect in the image from which it is made, so if you create another container you might need to install the package again if you want to use it in the newly created container
* you can also list the process running inside the container by running **docker top <container name or hash>** from outside the container
* there are two common ways of entering a container. The first method is entering the container at runtime. The second is entering or connecting to a container that is already running. In practice, the second method will be the one you'll use most often

**brief explanation of log files using dockers:**

* check the below command using 2 ports
* docker run -dit --name busylogs -p 8080:8080 -p 5000:5000 jenkins/jenkins:lts
* here notice Jenkins/Jenkins:lts ->Jenkins before forward slash is called userid or userid namespace or namespace , the Jenkins after the forward slash is the image name, lts stands for long term support
* to access above container from web browser you need password, password is unique and to fetch it we need to check logs
* if you want to see the runtime logs generated use
* docker logs -tf <container name>
* after each change that happens in browser will now be seen as log in the terminal
* to specify logs after a certain date use
* docker logs -t --since <yyyy-mm-dd> <container name>
* logs are created outside the container, inside the container, as well as the docker engine also created logs of its own, (docker sends its logs to its host machine)
* docker hub is the default image registry, the primary function of registry is to store images. Pull and push them from and to registry
* a repository can hold multiple images, and each image with the help of a tag, tag might denote the version of the image, they can also refer to different variations of an image.
* Dns name of docker hub registry is docker.io
* Pulling an image from docker registry:
* 1:docker pull docker.io/ubuntu:bionic
* 2:is same as docker pull ubuntu:bionic
* (command 1 and 2 are exactly same, i.e, even after running both commans you will have only one image)
* 3:docker pull registry.hub.docker.com/library/ubuntu:bionic – this command is pulling from the dns name: registry.hub.docker.com
* the above 1,3 commands are same, but the repository is different, both images will have same tags and same image id’s
* to create container out of the image use:
* ex:docker run -dit registry.hub.docker.com/library/ubuntu:bionic
* syntax: docker run <options> registryaddress/repository(or namespace)/image:tag
* if you don’t specify tag, latest tag will be used
* to remove the image created in above command 1:
* use docker -rm ubuntu:bionic (we have to specify the tag because, the tag used here Is not the latest one)
* **here even if the tag and image id are same, we are able to delete it using the repository name**
* docker images are prone to malware and images from unofficial registries are more prone, so if possible trying to isolate them can mitigate the risk
* different types of container registry is provided by the topmost companies for example Amazon provides ECR elastic container registry similarly Google provides GCR Google container registry and red hat linux provides quick QUAY
* If your images contain sensitive information you can make your docker private after that you might need to use credentials to access the images in the private docker

**Images:**

Small, portable,disposable

Contains only packages required to provide intended service

**Volumes:**

To persist or to save data generated or used by a container, we use volume

To share same data between multiple containers use volume

-v or --v are the parameters, but now –mount is pereferred in the place of -v or --volume

root@TIGER02143:~# docker volume --help

Usage: docker volume COMMAND

Manage volumes

Commands:

create Create a volume

inspect Display detailed information on one or more volumes

ls List volumes

prune Remove all unused local volumes

rm Remove one or more volumes

Run 'docker volume COMMAND --help' for more information on a command.

We can create,inspect, list, remove unused volumes or specific volumes as well

-oberve the below commands in which we create a container and assing some volume(storage space) to it

root@TIGER02143:~# docker volume create mydata1 (vreating a volume)

mydata1

root@TIGER02143:~# docker inspect mydata2

[]

Error: No such object: mydata2

root@TIGER02143:~# docker inspect mydata1 (checking the information related to the volume that is created)

[

{

"CreatedAt": "2022-04-14T11:09:00Z",

"Driver": "local",

"Labels": {},

"Mountpoint": "/var/lib/docker/volumes/mydata1/\_data", (all files related to this volume are stored here)

"Name": "mydata1",

"Options": {},

"Scope": "local"

}

]

Syntax to create docker and assign volume to it

Docker run -d --name <container name> --mount source =<volume name>, destination=/root/volumes <image name> (here the data stored in "/var/lib/docker/volumes/mydata1/\_data" aka mountpoint will also be available in the destination i.e, /root/volumes)

root@TIGER02143:~# docker run -t --name withvolume --mount source=mydata1, destination=/root/volumes nginx

if the volume mydata1 has some files "/var/lib/docker/volumes/mydata1/\_data" they will also be available in /root/volumes of withvolume container used above, if you now create another container and give access to the mydata1 volume, then also the files in the "/var/lib/docker/volumes/mydata1/\_data" will also be available In the destination of the container, and the container can have read,write (RW) access or read alone , if you have RW access files created in container with RW access will also reflect in volume files

to create a container with readonly use:

docker run -d --name <container name> --mount src=<path of volume>,dst=<path in the container where you want the files of volume to be copied>,readonly <image name>

if the volume is not there, docker creates it

we can also create ephemeral volume which will not last for long and are deleted as soon as container is removed

--mount is used when attaching a volume with the "docker run" command

note:

alias for source in above command:src

alias fro destination in above command: target,dst

* it is important to note aliases because some other people use them and we should be understanding if we encounter them
* if we delete container but not the volume associated with it, then the volume will be still there, if you want to delete it use **docker volume rm** command
* therefore volume is not deleted unless explicitly specified

networking through docker involves chapter 29-31, which I did not pay much attention, in case you might want to revisit them, if there is a need

brief points on network:

docker has networks and default network is bridge, if you don’t explicitly specify any network for a container, then bridge network is used for that container, containers that are connected to bridge network can communicate with each other directly.

-when you use **docker network inspect bridge,** you can see the containers with their id’s being allotted various resources such as different mac address,ipv4 and ipv6 addresses

-usecase of network: let us suppose there is a blog and you want to have two containers one for web server and another for database and you don’t want any other container access the database, then you can create a network for both the containers alone and don’t let any other container connect to the network, both the containers use different volumes and same network, when you don’t want to have port mentioned explicitly using -p, because you don’t want to expose it to public and only the web server should access it (creating a network is creating a bridge,before you create a database container, you have to create database)

**a container can be attached to multiple networks at a time, if network is not specified while a container is being created, by default it will be attached to bridge network.**

Create network using: **docker network create <name>**

Command:

Docker run –name <container name> –network <network> -p <port address> --mount src=<volume name> dst=<destination > -dit <image >:<tag>

Expl: a container is created and is told to use a specified network and a specified port with some specified volume which runs in detached mode using a particular image and a particular tag

-to check wether to containers (cont1,cont2) are connected by a network, check the ip address of cont1 and use exec command on cont2 using command **docker exec -it <cont2> bash it will open bash**, now the try connecting to cont1 ipv4 address using curl command: **curl -m 2 http:<ipv4 address>**

-**m 2** will make it timeout for 2 seconds

**DOCKER SWARM:** a cluster management and orchestration tool by docker, how to add nodes to the swarm and how to create services that run in the swarm

**Use of Orchestrators:**

* auto scale up: adds more containers when workload increases
* auto scale down: removes containers when workload decreases
* high availability – ensures enough containers are **running**  to provide the service
* garbage collection – terminates containers that have been crashed
* Docker swarm=swarm mode
* Provides high level of service availability by clustering docker host systems
* Node = a system that is part of docker swarm
* Node can be docker engines(docker hosts), managers, workers
* Replicas/scaling: additional containers to handle increasing work load
* Encrypts communication between nodes
* Did not dig deeper in to swarms futher, if needed go through chapter 32 in section 10

**Udemy course: Docker Essentials for Python Developers by chris selmer**

* Controllable and easily repeatable environments can be shared across team
* Speed of provisioning of complex environments
* Most of the cloud native apps based on microservices can run on containers
* A trained model can run on production environment using containers
* Kubernetes runs containers
* Docker is linux native software, grown from original linux containers
* There are some limitations related to GUI(tkinter,PyQT,openGL)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **.** | **windows** | **macos** | **unix** | **note** |
| webapps and microservices | yes | yes | yes |  |
| gui (tkinter,pyQt,OpenGL) | NO | NO | yes | need some external set up to make them acessible in windows and macos |
| Datascience with jupyter | yes | yes | yes |  |
| Datascience with anaconda/spyder | NO | NO | yes |  |
| general ML | yes | yes | yes | ML is really cool with docker |
| ML with gpu acceleration | NO | NO | yes |  |
| CV with webcam | NO | NO | yes | it is complicated and cumbersome to get access webcam with windows and macos |
| general python programming | yes | yes | yes |  |

Linux is a great choice to use python and it supports almost all distributios

**Mandatory docker components:**

**Docker runtime:** this executes containers and creates images, it also communicates with dockerhub and other container registries

It does all the jobs such as pulling images from registry and so on..(just like compiler does all the work for us)

**Docker command:**  It is cli, it manages all aspects of containers and **docker runtime** as well

**Docker compose:**

Docker Compose is used **to run multiple containers as a single service**. For example, suppose you had an application which required NGNIX and MySQL, you could create one file which would start both the containers as a service without the need to start each one separately.

* Docker Compose is a quality of life improvement tool over the Docker CLI but still uses the Docker CLI under the hood. It lets you declare Docker CLI commands in the form of YAML and has its own CLI which lets you easily manage 1 or more Docker containers.
* Use case:
* It lets you quickly spin up and destroy 1 or many containers. It is what you’ll use in your day to day in development, CI and in some cases even production.
* I say “some cases” for production because while Docker Compose is excellent for single server deploys, it’s not really meant for multi-host deploys, however the tool that is aimed at multi-host deploys uses a compatible version of Docker Compose’s YAML file.

If you want to learn how to use the Docker CLI and Docker Compose by example then check out the Dive Into Docker course . It will take you from ground 0 all the way to comfortably using Docker on your own projects. Docker Compose is a quality of life improvement tool over the Docker CLI but still uses the Docker CLI under the hood. It lets you declare Docker CLI commands in the form of YAML and has its own CLI which lets you easily manage 1 or more Docker containers.

**Use case:**  
It lets you quickly spin up and destroy 1 or many containers. It is what you’ll use in your day to day in development, CI and in some cases even production.

I say “some cases” for production because while Docker Compose is excellent for single server deploys, it’s not really meant for multi-host deploys, however the tool that is aimed at multi-host deploys uses a compatible version of Docker Compose’s YAML file.

If you want to learn how to use the Docker CLI and Docker Compose by example then check out the [Dive Into Docker course](https://diveintodocker.com/) . It will take you from ground 0 all the way to comfortably using Docker on your own projects.

By default docker containers are executed with super user access, that means it can literally do anything (they directly run on linux kernel), and because of this we should be careful with the container images from unknown sources, but it does not guarantee that container images from docker registry are fully safe, but they are better

To invoke python using image from docker you can use the below command:

* docker run -it --name mypython1 -p 5000:5000 -p 8000:8000 -v ${PWD}:/app python:3.7 bash

the above command is creating a container named mypython1 with app as its folder where it can read or write files, the python:3.7 specifies it need to have **python image with 3.7** version, **bash** will open a bash shell,

the python official image has pip,python library and all system libraries installed,

after executing above command it will open bash and then if you type python, an python interpreter of version3.7 is activated