How to Docker #1

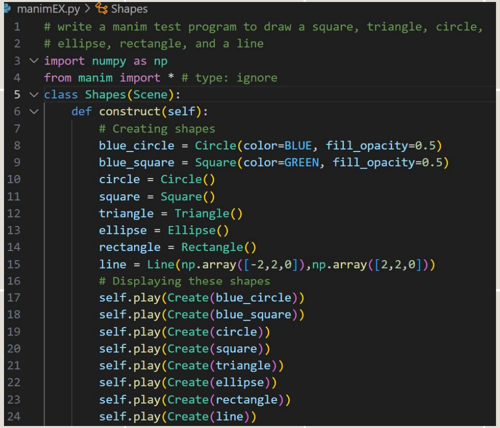
Made for May 21st 2025 By Jacob Lazarchik

Overview

1. **Why Docker?** - Problems with legacy lab code & motivation to containerize
2. **What is Docker?** — Concepts of containers, images, and the Docker engine
3. **Building Docker Images** — Dockerfiles, offline dependency setup, and structure
4. **Running & Using Docker** — GUI vs CLI, Docker Compose, device passthrough
5. **Lab Integration** — USB hardware control, GUI deployment, cross-platform support
6. **Goals & Roadmap** - Plans for the next “How to Docker”

Why Docker?

Your code



manimEX.py > Shapes

```python
# write a manim test program to draw a square, triangle, circle,
# ellipse, rectangle, and a line
import numpy as np
from manim import * # type: ignore
class Shapes(Scene):
    def construct(self):
        # Creating shapes
        blue_circle = Circle(color=BLUE, fill_opacity=0.5)
        blue_square = Square(color=GREEN, fill_opacity=0.5)
        circle = Circle()
        square = Square()
        triangle = Triangle()
        ellipse = Ellipse()
        rectangle = Rectangle()
        line = Line(np.array([-2,2,0]),np.array([2,2,0]))
        # Displaying these shapes
        self.play(Create(blue_circle))
        self.play(Create(blue_square))
        self.play(Create(circle))
        self.play(Create(square))
        self.play(Create(triangle))
        self.play(Create(ellipse))
        self.play(Create(rectangle))
        self.play(Create(line))
```

-5+ Years
-No Requirements.txt
-Online Dependencies
-Aging 1990s-Style GUI
-2M local file imports
-Single .venv (if it exists)
-No Readme.txt

-1 major Python release behind
-Nonexistent comments
-Only magic numbers
-Written by a freshman

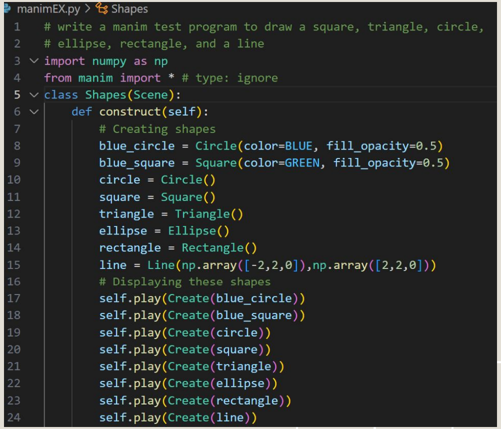
Trash Fire

Goal of this Project

Trash Fire

No more putting out fires.

New code



```python
# write a manim test program to draw a square, triangle, circle,
# ellipse, rectangle, and a line
import numpy as np
from manim import * # type: ignore
class Shapes(Scene):
    def construct(self):
        # Creating shapes
        blue_circle = Circle(color=BLUE, fill_opacity=0.5)
        blue_square = Square(color=GREEN, fill_opacity=0.5)
        circle = Circle()
        square = Square()
        triangle = Triangle()
        ellipse = Ellipse()
        rectangle = Rectangle()
        line = Line(np.array([-2,2,0]),np.array([2,2,0]))
        # Displaying these shapes
        self.play(Create(blue_circle))
        self.play(Create(blue_square))
        self.play(Create(circle))
        self.play(Create(square))
        self.play(Create(triangle))
        self.play(Create(ellipse))
        self.play(Create(rectangle))
        self.play(Create(line))
```

Docker

Containerize and Forget.

Benefits of Docker

Unified Lab UI

One image, one UI – PyQt front-end + REST API compiled into a single container that renders locally or over VNC/WebRTC.

3 Operating Systems, 1 UI

Instant student on-boarding – clone repo → docker compose up → fully-featured GUI talking to real hardware in < 15 min.

<15 min Setup Time

Reproducibility

Tagged image per experiment – every published figure or dataset links back to the Docker image + exact controller configuration.

1 Image Tag / Published Figure

Containerized stack – anyone can rebuild the exact environment, even 5 years from now.

100% Reproducible Experiments

Plug-and-play Hardware

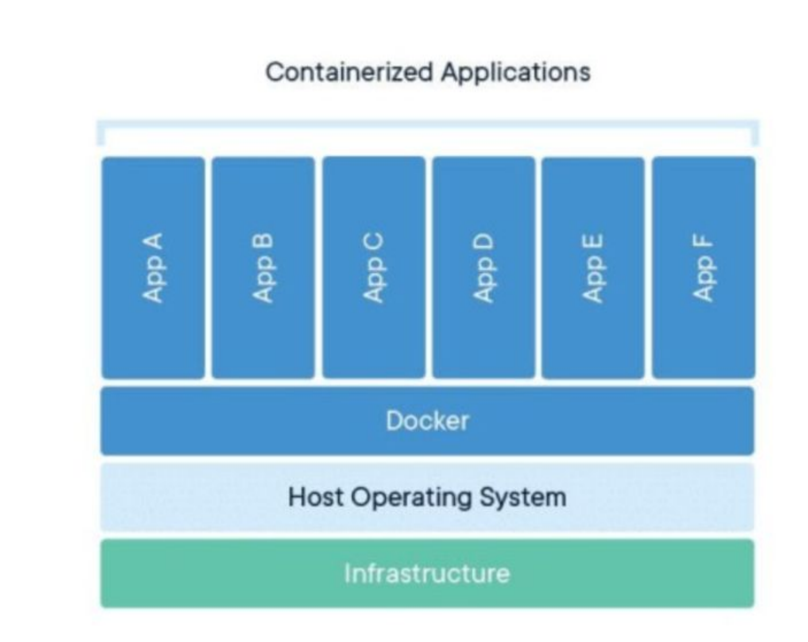
Zero-conf USB mapping – container ships udev rules & Thorlabs SDK; plug in a TDC001, K-cube, or ESR-PRO and the GUI auto-detects.

0 Manual Installs

Python 3.7, NumPy 1.20, custom USB driver — always same versions, even 5 years from now.

No more dependency rot.

What is Docker? - Containers



1. Docker is *NOT* a virtual machine host (hypervisor)

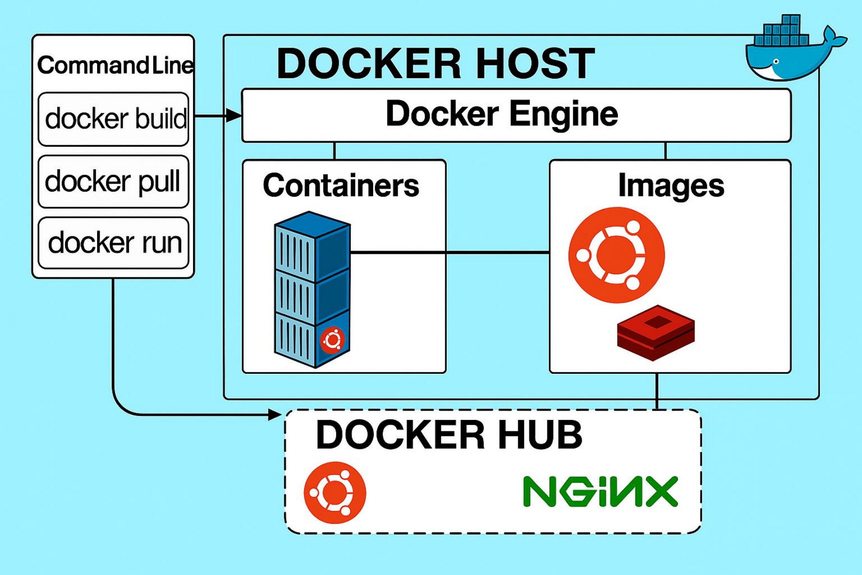
2. Docker "Containers" run with basically native speeds on all Operating Systems (Windows, MacOS, Linux), *just like your original python code.*

3. The Docker "Engine" uses the host OS and its resources (kernel) to run, *just like any other program.* Docker is simply a compatibility layer/sandbox for your code. (Google Chrome and many other browsers operate this way)

4. However, *for simplicity* you can still think of a Docker "Container" as separate computer - with its own IP/Sockets/Address, OS, Command Line, File Directories, Software, Network, and Python Code.

5. *To clarify, "containers" are not actually virtual machines! They share a host kernel.*

What is Docker? - Images



1. Previously, I stated a Docker “Container” has its own OS+Python Code+Dependencies+etc.

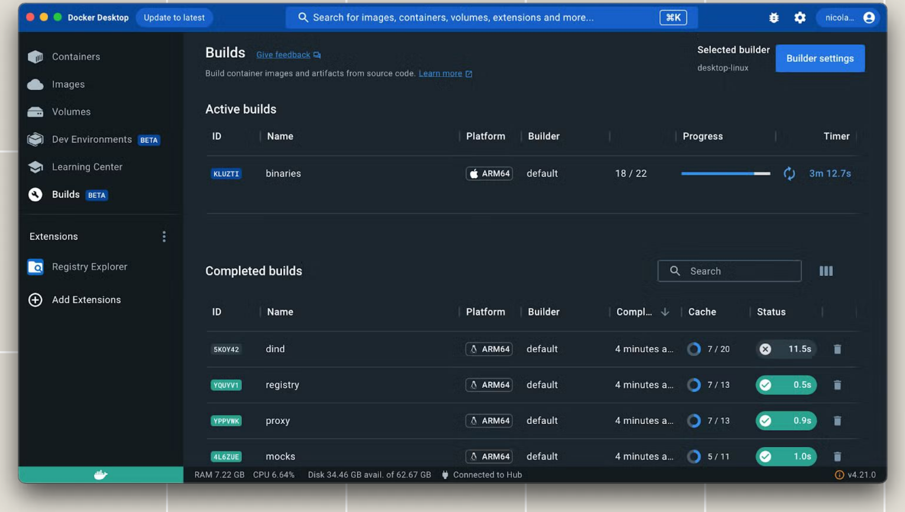
2. All this “stuff” inside the “Container” is actually stored in a single file called a Docker “Image”

3. *Docker “Containers” run Docker “Image” files.* **“**Images” can be made yourself (by ‘building’ an image) or imported from an online “registry” like “Docker Hub” *and customized*.

4. The “customization” part of one image is done with two very simple text files: A “Dockerfile”, and a “Docker Compose” file; Then you “build” the “image” again. This new “image” is stored in your computer’s local “registry”.

5. The “Docker Hub” *registry* has 100,000+ free Docker “Images” that you can run inside Docker “Containers” by default; For example: PostgreSQL, Python-3.11, etc.

Interaction - GUI or CLI?



Builds

Build container images and artifacts from source code.

Selected builder
desktop-linux

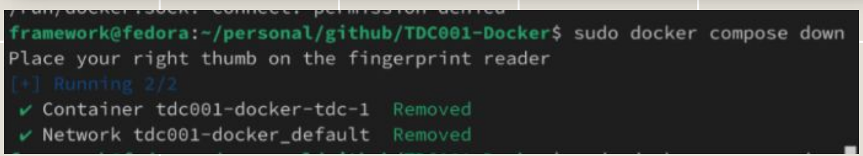
Active builds

ID	Name	Platform	Builder		Progress	Timer
KLUZTI	binaries	ARM64	default	18 / 22		3m 12.7s

Completed builds

ID	Name	Platform	Builder	Compl...	Cache	Status
5KOY42	dind	ARM64	default	4 minutes a...	7 / 20	☒ 11.5s
YQUYV1	registry	ARM64	default	4 minutes a...	7 / 13	✓ 0.5s
YPPVWK	proxy	ARM64	default	4 minutes a...	7 / 13	✓ 0.9s
4L6ZUE	mocks	ARM64	default	4 minutes a...	5 / 11	✓ 1.0s

RAM 7.22 GB CPU 6.64% Disk 34.46 GB avail. of 62.67 GB Connected to Hub v4.21.0



```
/run/docker.sock: connect: permission denied
framework@fedora:~/personal/github/TDC001-Docker$ sudo docker compose down
Place your right thumb on the fingerprint reader
[+] Running 2/2
 ✔ Container tdc001-docker-tdc-1  Removed
 ✔ Network tdc001-docker_default  Removed
```

1. The Docker "Engine" has two ways of interacting with it (clients)

2. "Docker Desktop" (Graphical User Interface) + "Docker CLI" (Command Line Interface) are two "clients" you can use

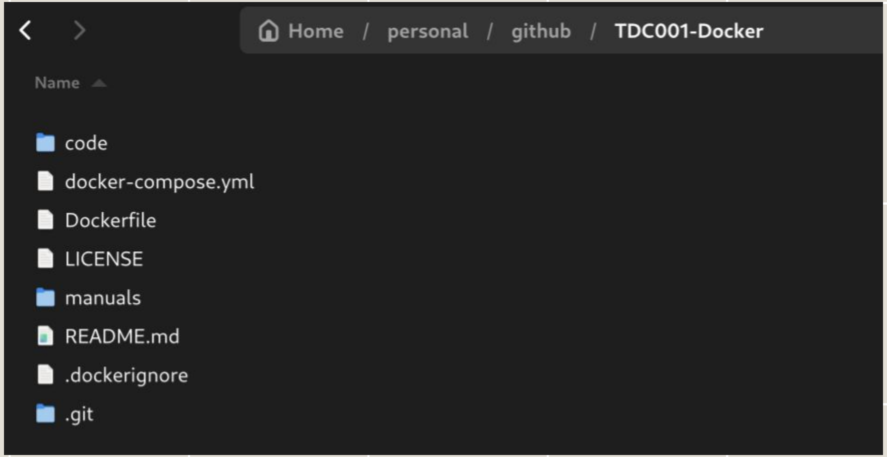
3. The command line interface (CLI) is the least frustrating and quickest usually

4. The GUI allows you to quickly access the terminal (or command line/shell) of Docker "containers" as you would if you made a standard "print()" and "input()" in your Python code. Also has a simple view to see what "containers" are running and the "images" in your local "registry".

5. To do this in CLI, you must enter "docker compose exec tdc bash" (or something similar), which then shows you the "print()" and others, or you use "ENTRYPOINT" (covered later).

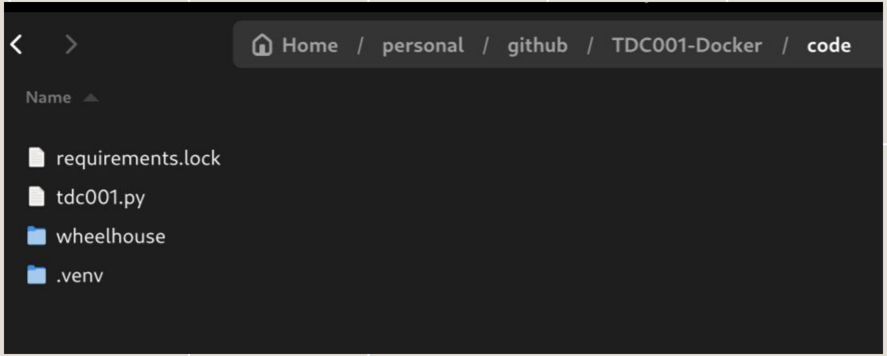
6. *However, Downsides to Desktop exist...*

My Directory


Home / personal / github / TDC001-Docker

Name

- code
- docker-compose.yml
- Dockerfile
- LICENSE
- manuals
- README.md
- .dockerignore
- .git


Home / personal / github / TDC001-Docker / code

Name

- requirements.lock
- tdc001.py
- wheelhouse
- .venv

1. In most cases, start by writing your python code *as you normally would!* No tricks here.

2. *Ensure your code is designed for operating only one device or purpose. "One service per container" is the best way to avoid problems in the future.*

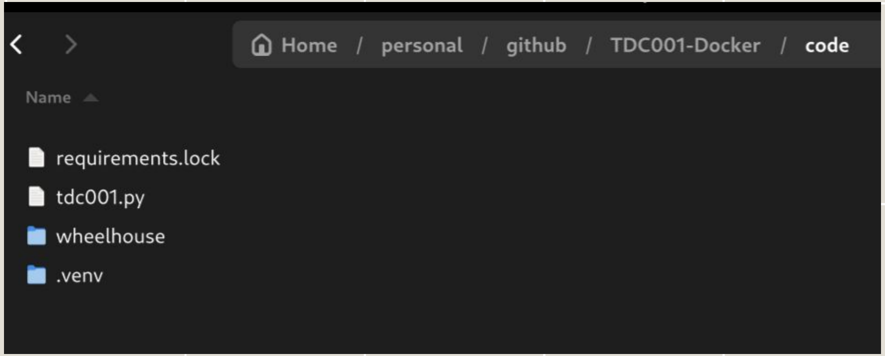
3. If you want one container (not multiple interacting), you only need a "Dockerfile" of a few lines.

4. For more *advanced* setups - convenience, many "containers", networking, and other capabilities - you must include a "docker-compose.yml" file.

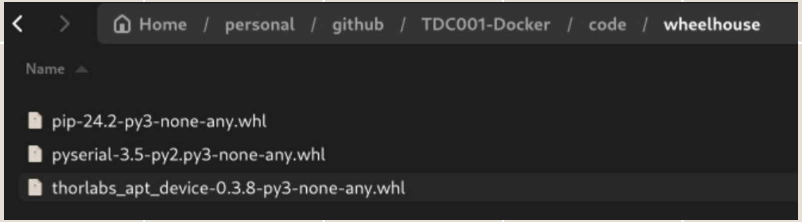
5. You run your "docker build" command, and then an "image" appears in Docker Desktop under "images" , or in your local offline "registry".

6. This 'built' "image" can now be run using "docker compose up", "docker run", or GUI. Also, it can be shared to others with Docker installed.

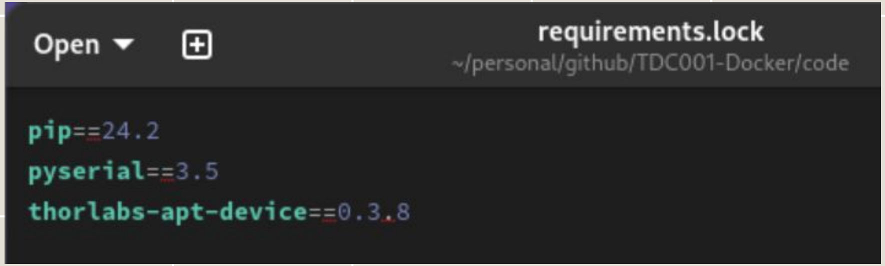
Critical - Wheelhouse + Lockfile



requirements.lock
tdc001.py
wheelhouse
.venv


Home / personal / github / TDC001-Docker / code / wheelhouse
Name

pip-24.2-py3-none-any.whl
pyserial-3.5-py2.py3-none-any.whl
thorlabs_apt_device-0.3.8-py3-none-any.whl



```
pip==24.2
pyserial==3.5
thorlabs-apt-device==0.3.8
```

1. Freeze your dependencies and store them locally so your Docker “builds” don't rely on online servers! This ensures others can “build” a new “image” from your github repo—even years later—without broken links or missing packages. Versions of certain packages are not available from online registries forever!

2. In your virtual environment (.venv) CLI or something, run “pip freeze --all > requirements.lock”. The “requirements.lock” file is just a text file of your downloaded packages and their versions that are used by your code currently. Basically, you are “freezing” your code dependencies at that point in time.

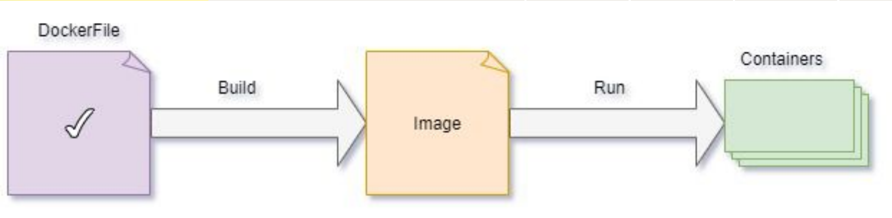
3. create a “/wheelhouse” directory (or “mkdir -p wheelhouse”) (run this once)

4. “pip download --dest=wheelhouse -r requirements.lock” simply downloads the full “wheel” (.whl) files into this directory for you to download these packages offline in the future during the “build” process. Explained Later.

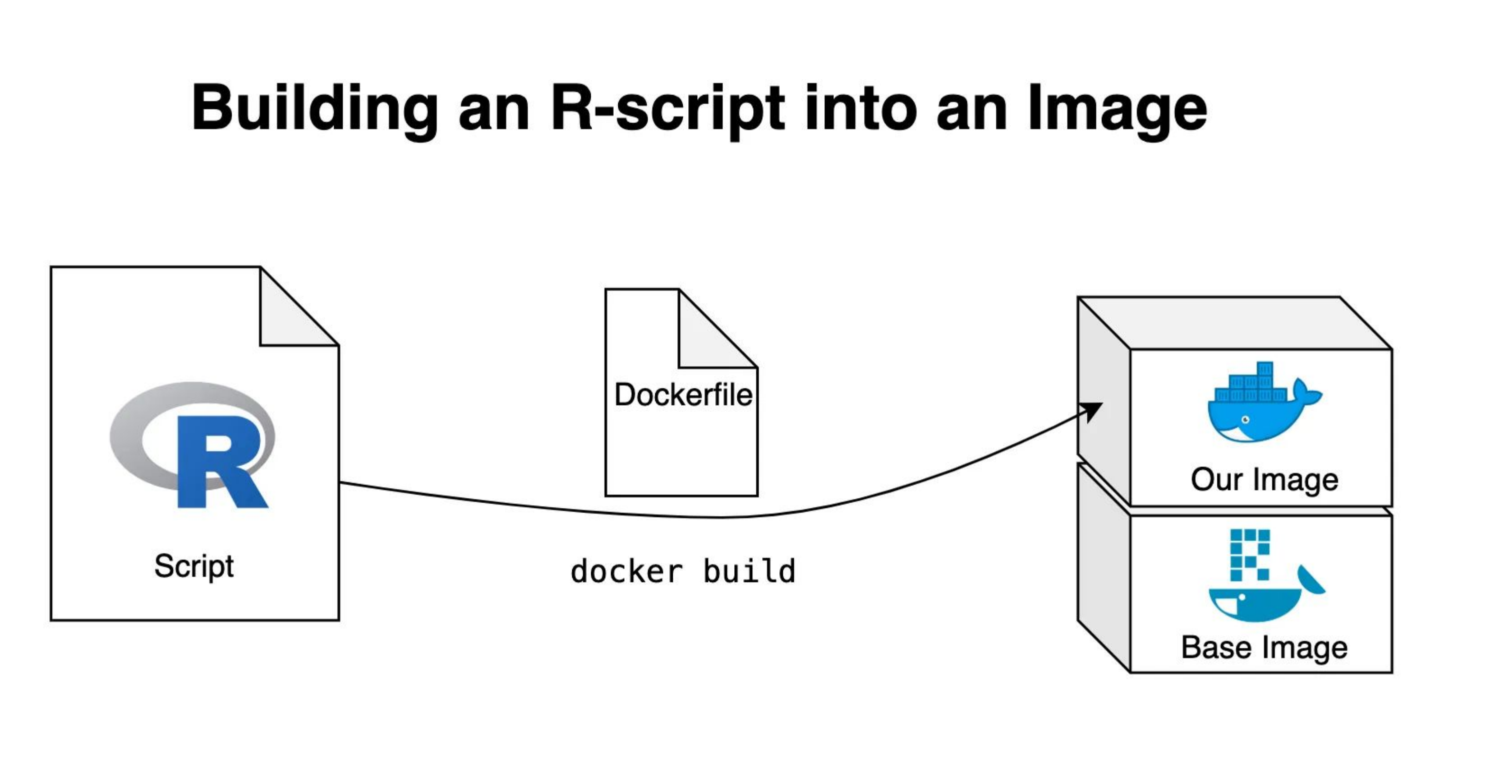
Importance

This forces lab members to properly store code dependencies from the beginning.

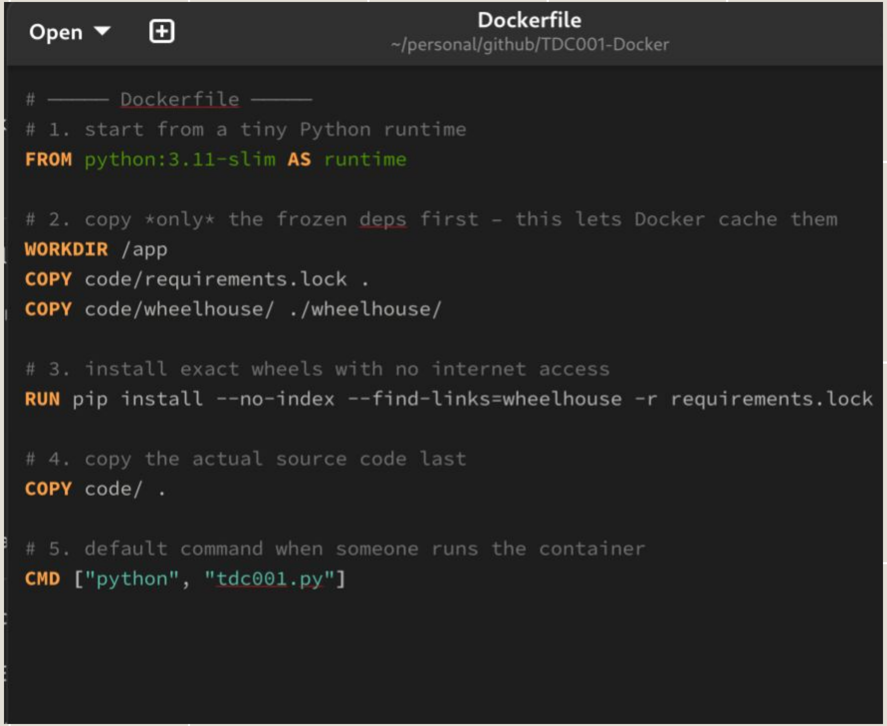
By structuring your “Dockerfile” as shown in future slides, “pip” installs strictly from your local “/wheelhouse” during the “docker build” “image” creation. This reminds you to regenerate your “/wheelhouse” and “requirements.lock” each time you run “docker build” (otherwise your “image” may be missing dependencies and throw an error while running in a “container”). (Explained Later in more clarity)



Building Images Illustrated



The Dockerfile - Beginning



```
# ───── Dockerfile ─────
# 1. start from a tiny Python runtime
FROM python:3.11-slim AS runtime

# 2. copy *only* the frozen deps first - this lets Docker cache them
WORKDIR /app
COPY code/requirements.lock .
COPY code/wheelhouse/ ./wheelhouse/

# 3. install exact wheels with no internet access
RUN pip install --no-index --find-links=wheelhouse -r requirements.lock

# 4. copy the actual source code last
COPY code/ .

# 5. default command when someone runs the container
CMD ["python", "tdc001.py"]
```

1. The Dockerfile is run once during “docker build”, this constructs our Docker “image”

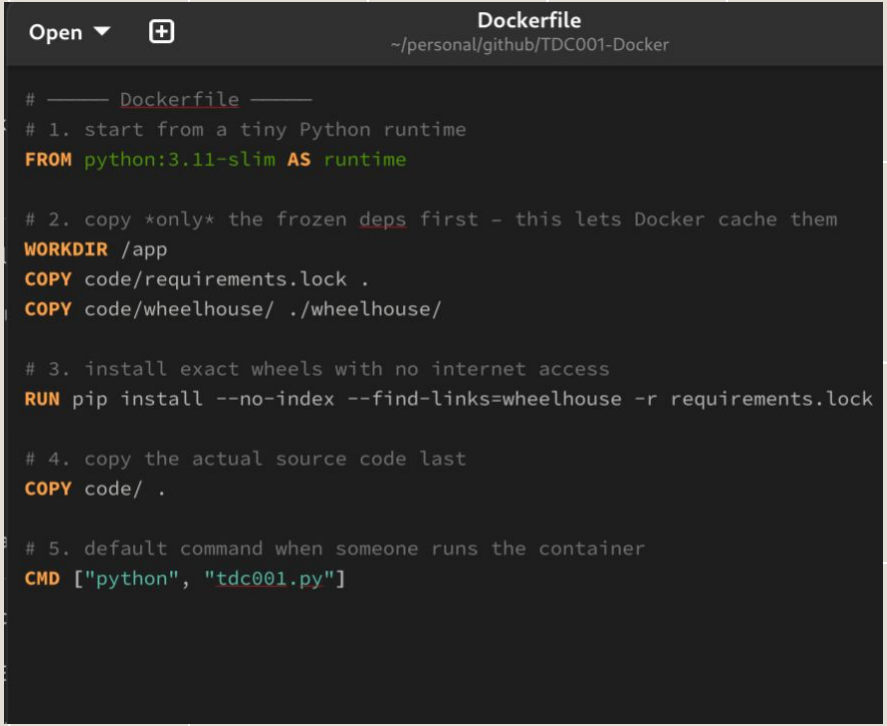
2. The First Line, importing a full python 3.11 Docker “Image” from the “Docker Hub” (pre-made images) registry.

3. The Second Line is the same as “cd /app”, basically setting our “Working Directory” *inside the image’s filesystem*

4. The Third Line copies our “requirements.lock” file. This lists all python dependencies (pyserial, pyvisa, etc) in plain text w. version number. The “ .” at the end means it saves in “/app” or the “Working Directory” set above.

5. The Fourth Line copies my original “wheelhouse/” directory (stores “zipped” versions of package installs you’d normally get online). Unlike the previous line, the “ ./wheelhouse/” infers it saves the “zipped” packages to “/app/wheelhouse” .

Offline Pip Installs



```
# ———— Dockerfile ————
# 1. start from a tiny Python runtime
FROM python:3.11-slim AS runtime

# 2. copy *only* the frozen deps first – this lets Docker cache them
WORKDIR /app
COPY code/requirements.lock .
COPY code/wheelhouse/ ./wheelhouse/

# 3. install exact wheels with no internet access
RUN pip install --no-index --find-links=wheelhouse -r requirements.lock

# 4. copy the actual source code last
COPY code/ .

# 5. default command when someone runs the container
CMD ["python", "tdc001.py"]
```

1. The Fifth Line runs pip just like you would in the terminal, *but internally w/o internet...*

2. First, “no-index” means “do not contact the internet to download packages”.

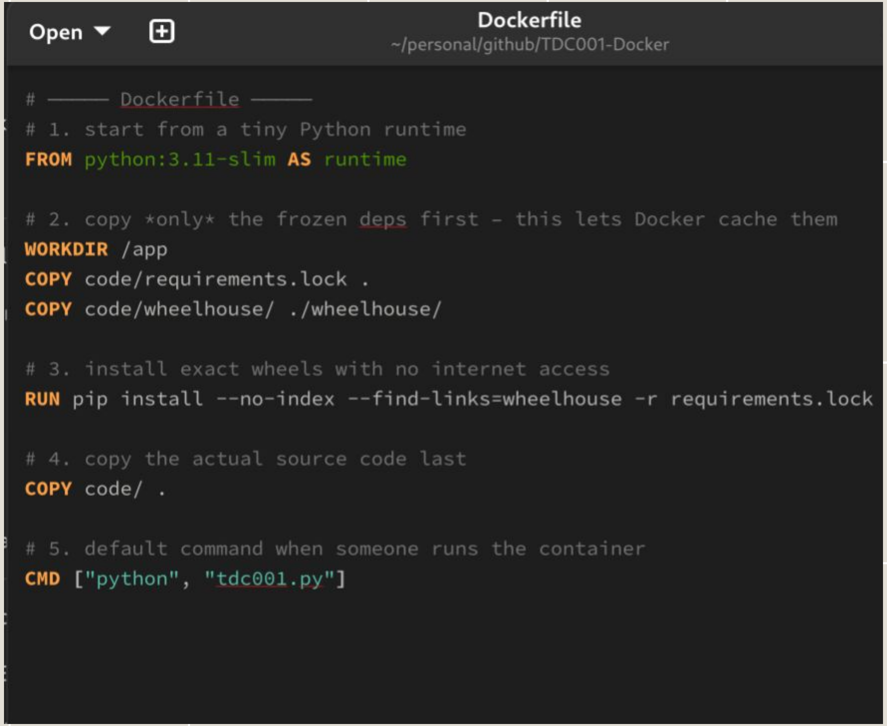
3. Second, “find-links=wheelhouse” means “go into the wheelhouse directory, this is your offline package registry to download dependencies” .

4. Third, “-r requirements.lock” means “read from the requirements.lock file and download the dependencies listed”.

5. *Due to the “no-index” + ”find-links”, pip only downloads from “zipped” packages put inside the new “/app/wheelhouse/” directory*

6. *By using wheelhouse + requirements + docker, we eliminate all future dependency issues indefinitely.*

Last Dockerfile Lines



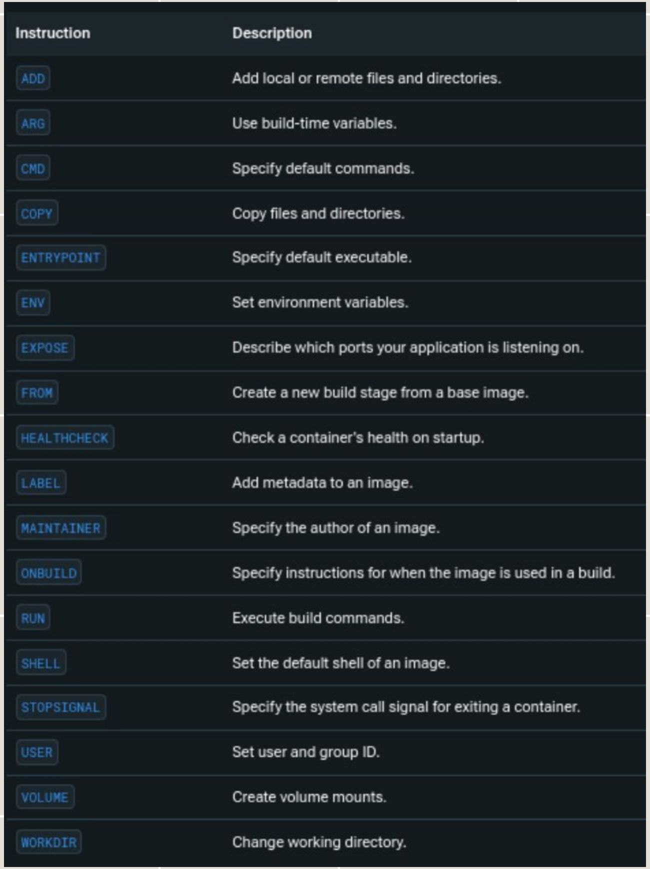
```
# ——— Dockerfile ———
# 1. start from a tiny Python runtime
FROM python:3.11-slim AS runtime

# 2. copy *only* the frozen deps first – this lets Docker cache them
WORKDIR /app
COPY code/requirements.lock .
COPY code/wheelhouse/ ./wheelhouse/

# 3. install exact wheels with no internet access
RUN pip install --no-index --find-links=wheelhouse -r requirements.lock

# 4. copy the actual source code last
COPY code/ .

# 5. default command when someone runs the container
CMD ["python", "tdc001.py"]
```

1. The Sixth Line (‘COPY code/ .’) copies *mostly* everything inside my original “code/” directory on my host machine to the “image”. Like previously, the “ .” following the end of this line means it saves to the working directory or “/app” we set above. Mostly, just tdc001.py is what this is for.

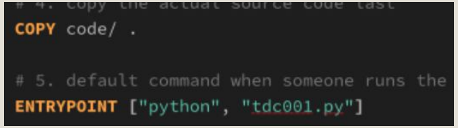
2. The reason I said *mostly* is because of the included “.dockerignore” file, which ignores the “/code/.venv/” directory (virtual environment for python) and a few other files.

3. The Final Line (CMD) is different than “RUN” before. “CMD” is used for commands you wish to execute when the Docker “Container” starts (‘docker run’). “RUN” is for commands you want to execute while the “image” is being “built”. “CMD” runs my script like I did originally in command line “python tdc001.py”

Dockerfile - Reference

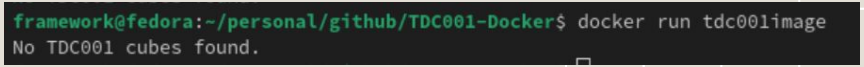


Instruction	Description
ADD	Add local or remote files and directories.
ARG	Use build-time variables.
CMD	Specify default commands.
COPY	Copy files and directories.
ENTRYPOINT	Specify default executable.
ENV	Set environment variables.
EXPOSE	Describe which ports your application is listening on.
FROM	Create a new build stage from a base image.
HEALTHCHECK	Check a container's health on startup.
LABEL	Add metadata to an image.
MAINTAINER	Specify the author of an image.
ONBUILD	Specify instructions for when the image is used in a build.
RUN	Execute build commands.
SHELL	Set the default shell of an image.
STOPSIGNAL	Specify the system call signal for exiting a container.
USER	Set user and group ID.
VOLUME	Create volume mounts.
WORKDIR	Change working directory.



```
# 4. copy the actual source code last
COPY code/ .

# 5. default command when someone runs the
ENTRYPOINT ["python", "tdc001.py"]
```

1. These are all the rest of the possible instructions you can include in a "Dockerfile", some I didn't use.(https://docs.docker.com/reference/dockerfile/)
2. Notable ones include "ENTRYPOINT", which is actually superior vs "CMD", really useful for Command Line Tools. Always enforces a script running vs CMD which doesn't always do this. Also makes sure any "print()" or "input()" statements are seen through the host terminal, just like running a normal python file:



```
framework@fedora:~/personal/github/TDC001-Docker$ docker run tdc001image
No TDC001 cubes found.
```

3. "ADD" is another useful one (similarly "VOLUMES"), to access a local or remote file from outside a container, persistent data storage.
4. EXPOSE for networking(doesn't actually open a port but people who receive image know how to communicate with the container).

Building and Running - Problems Arise

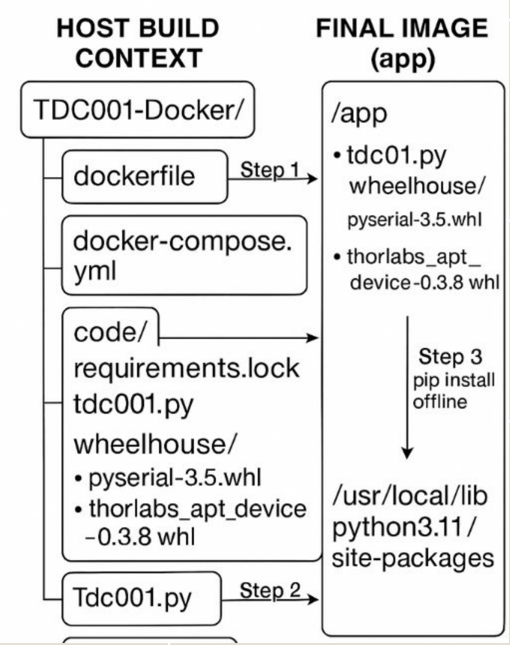
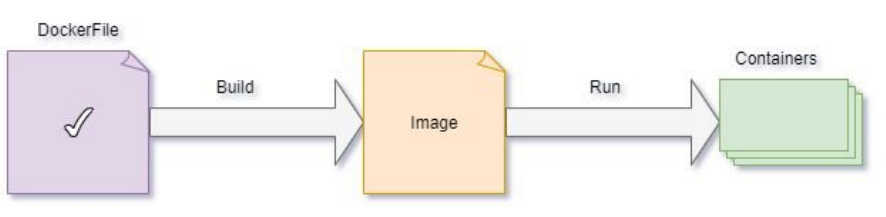
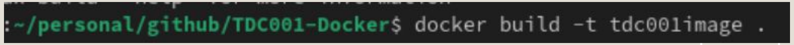


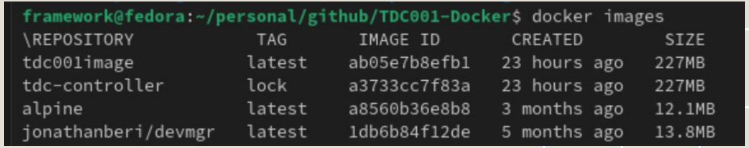
image made by chatgpt - partially wrong lol, but the concept is kinda there



1. At this point, you could immediately run a single simple script
2. “docker build -t image-name build-context”
   a. “image-name” is whatever you like
   b. “build-context” is the location of your “Dockerfile”
   c. it is best to cd into the directory of the Dockerfile you please and then you can just say the build context is “.”
   d. 

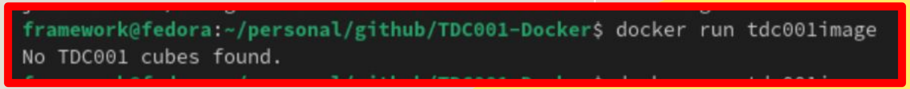
```
:~/personal/github/TDC001-Docker$ docker build -t tdc001image .
```

3. To see all your images, run “docker images”



```
framework@fedora:~/personal/github/TDC001-Docker$ docker images
\REPOSITORY             TAG       IMAGE ID       CREATED        SIZE
tdc001image            latest    ab05e7b8efb1   23 hours ago   227MB
tdc-controller         lock      a3733cc7f83a   23 hours ago   227MB
alpine                 latest    a8560b36e8b8   3 months ago   12.1MB
jonathanberi/devmgr    latest    1db6b84f12de   5 months ago   13.8MB
```

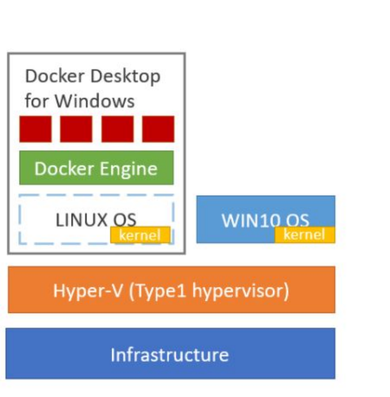
4. Now simply run “docker run image-name”:



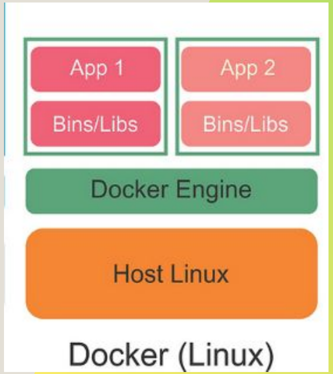
```
framework@fedora:~/personal/github/TDC001-Docker$ docker run tdc001image
No TDC001 cubes found.
```

5. **For USB devices, outside of Linux, this will not work (as seen above), due to device permissions!**

Windows Breaks USB Devices *Slightly*



How docker works in Windows



How docker works in Linux natively - note no virtualization

Why Docker works differently on Windows vs. Linux (and what I'm doing about it):

Docker was originally designed for Linux, so if you're using Linux, Docker can directly access your computer's hardware — including USB devices — with no extra setup. Everything just works out of the box.

But on Windows and macOS, Docker can't run directly on the system. Instead, it runs inside a small Linux virtual machine (VM). This VM is lightweight and fast (uses less than 0.2% of your CPU), and on Windows it runs through WSL (Windows Subsystem for Linux). WSL is very efficient, almost like running Linux directly on hardware.

However, here's the catch:

Since Docker is running inside a VM on Windows, it can't see USB devices (like lab hardware) unless you manually pass those devices into the virtual machine. That step takes extra work — and it's why Docker doesn't behave the same on Windows as it does on Linux.

Remaining Roadmap

Get Windows Up

-Finish server setup

-Download Docker Desktop

-clone my git repo

Get container for USB devices

-Docker documentation has a way of passing all USB devices to a container to expose ports again at the kernel level.

Get Existing code working perfectly

-make sure USB devices set up with minimal difficulty in a repeatable way

Set up Network via Docker Compose

- build and compose 3 containers, one for a GUI, our existing tdc001 controller, and the one for usb passthrough.

Use REST API or other

-get the two containers communicating in the easiest way possible with low latency

Goal

-by the end, this will be platform agnostic, native performance, and serve as a template for all future projects

Any questions?

Sources

https://docs.docker.com/get-started/ (introduction material)
https://docs.docker.com/manuals/ (for docker desktop documentation, install, etc)
https://docs.docker.com/reference/ (has everything for docker compose, CLI, etc)
https://thorlabs-apt-device.readthedocs.io/en/latest/gettingstarted.html (thorlabs info)
https://github.com/JacobGitz/TDC001-Docker (my git repo, most recent on dev branch)
https://pyserial.readthedocs.io/en/latest/tools.html (pyserial)

Other Useful Links
https://github.com/jesseduffield/lazygit
https://www.lazyvim.org/