

Docker

Intro to Docker - Virtual Machines

Virtual machines (VMs) are created through a process called virtualisation.

Virtualisation is a technology that allows you to create multiple simulated environments or virtual versions of something, such as an operating system, a server, storage, or a network, on a single physical machine.

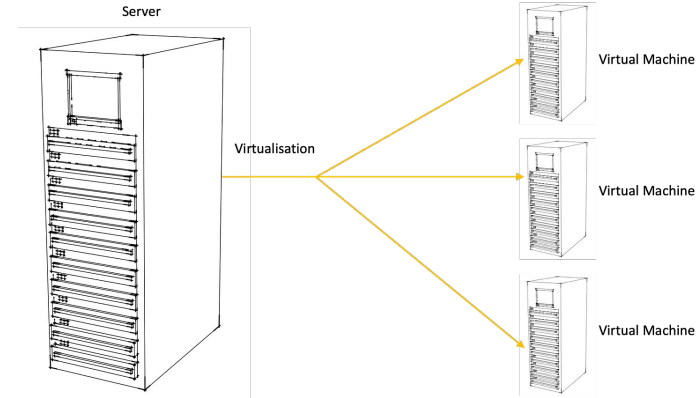
These virtual environments behave as if they are independent, separate entities, even though they share the resources of the underlying physical system.

Intro to Docker - Virtual Machines

Virtualisation allows a single physical computer or server to appear as multiple virtual machines (VMs), each with its own operating system and resources.

VMs virtualise the hardware. This simply means that a VM takes a single piece of hardware – a server – and creates virtual versions of other servers running their own operating systems. Physically, it is just a single piece of hardware.

Logically, multiple virtual machines can run on top of a single piece of hardware. This is essentially one or more computers running within a computer, as shown below.

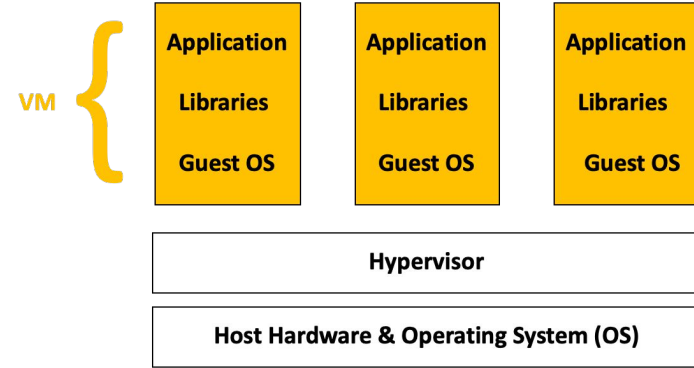


How does virtualisation works?

At the base, you have the host hardware and OS.

This is the physical machine that is used to create the virtual machines.

On top of this, you have the hypervisor. This allows multiple virtual machines, each with their own operating systems (OS), to run on a single physical server.

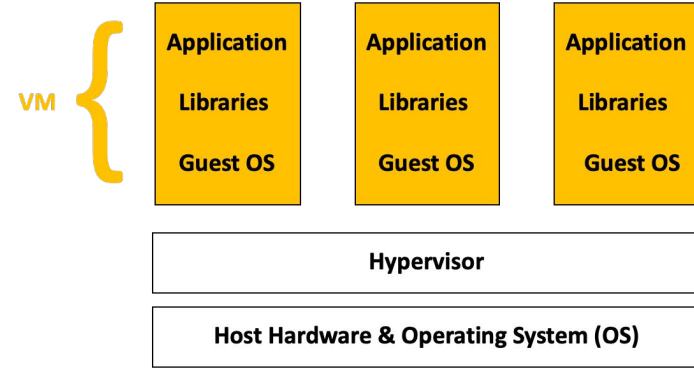


How does virtualisation works?

VMs have a few downsides, though, which containers address.

Two downsides particularly stand out:

1. **VMs consume more resources:** VMs have a higher resource overhead due to the need to run a full OS instance for each VM. This can lead to larger memory and storage consumption. This in turn can have a negative effect on performance and startup times of the virtual machine.

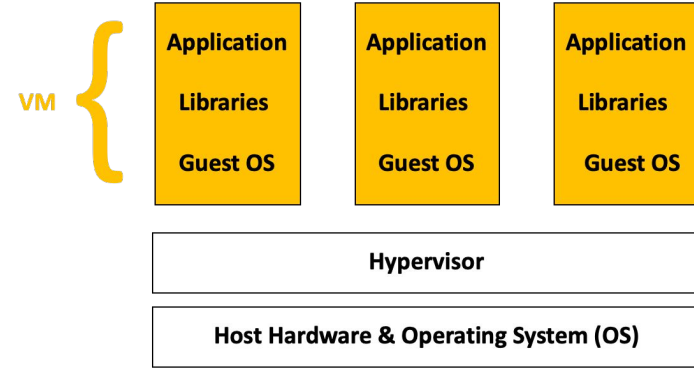


How does virtualisation works?

VMs have a few downsides, though, which containers address.

Two downsides particularly stand out:

2. Portability: **VMs are typically less portable** due to differences in underlying OS environments. Moving VMs between different hypervisors or cloud providers can be more complex.

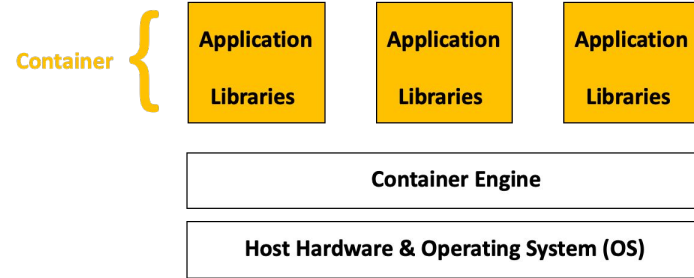


How does Container works?

A container is a lightweight, standalone, and executable software package that includes everything needed to run a piece of software, including the code, runtime, system tools, and libraries.

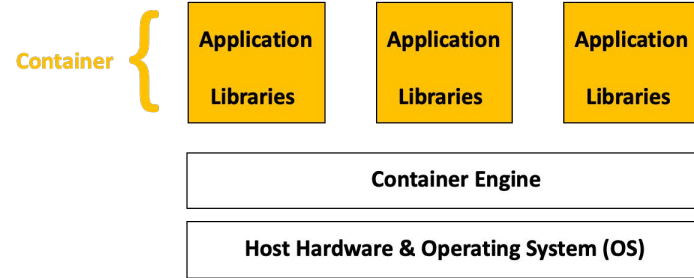
Containers are designed to isolate applications and their dependencies, ensuring that they can run consistently across different environments.

Whether the application is running from your computer or in the cloud, the application behaviour remains the same.



How does Container works?

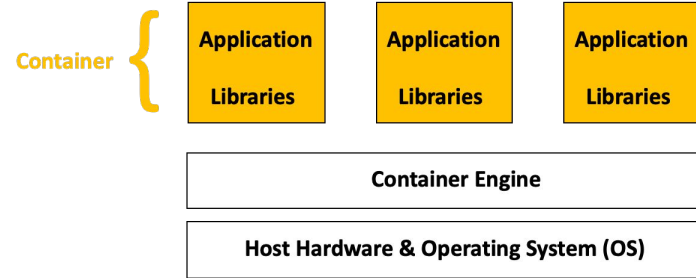
Unlike VMs which virtualise the hardware, [containers virtualise the operating system](#). This simply means that a container uses a single OS to create a virtual application and its libraries. Containers run on top of a shared OS provided by the host system.



How does Container works?

The container engine allows you to spin up containers.

It provides the tools and services necessary for building, running, and deploying containerised applications.



How does Container works?

Containers have several benefits:

1. **Portability:** Containers are designed to be platform-independent. They can run on any system that supports the container runtime, such as Docker, regardless of the underlying operating system. This makes it easier to move applications between different environments, including local development machines, testing servers, and different cloud platforms.

How does Container works?

Containers have several benefits:

2. **Efficiency:** Containers share the host system's operating system, which reduces the overhead of running a virtual machine with multiple operating systems. This leads to more efficient resource utilization and allows for a higher density of applications that can run on a single host.

How does Container works?

Containers have several benefits:

3. **Consistency:** Containers package all the necessary components, including the application code, runtime, libraries, and dependencies, into a single unit. This eliminates the "it works on my machine" problem and ensures that the application runs consistently across different environments, from development to production.

How does Container works?

Containers have several benefits:

4. **Isolation:** Containers provide a lightweight and isolated environment for running applications. Each container encapsulates the application and its dependencies, ensuring that they do not interfere with each other. This isolation helps prevent conflicts and ensures consistent behaviour across different environments.

How does Container works?

Containers have several benefits:

5. **Fast Deployment:** Containers can be created and started quickly, often in a matter of seconds. This rapid deployment speed is particularly beneficial for applications that need to rapidly scale up or down based on demand.

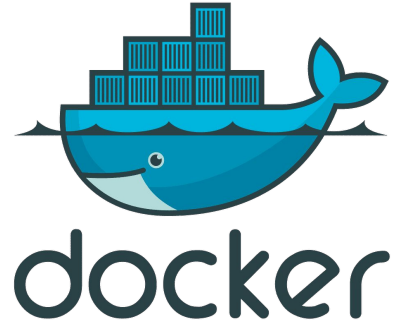
What is Docker?

Now that we have covered VMs and containers, what exactly is Docker?

Docker is simply a tool for creating and managing containers.

Docker has two concepts that are useful to understand:

- Dockerfile
- Docker Images.



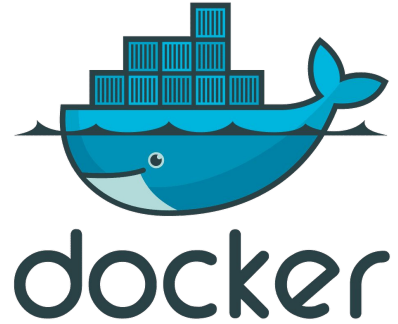
What is Docker? - Dockerfile and Docker Images

A **Dockerfile** contains the set of instructions for building a Docker Image.

A **Docker Image** serves as a template for creating Docker containers.

It contains all the necessary code, runtime, system tools, libraries, and settings required to run a software application.

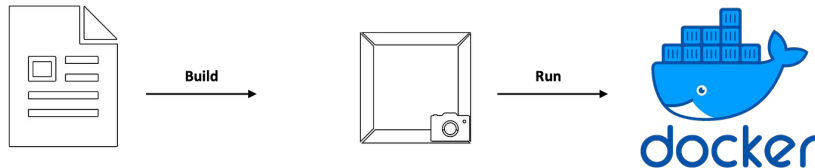
So, a **Dockerfile** is used to build a **Docker Image** which is then used as the template for creating one or more **Docker containers**.



1. Dockerfile

2. Docker Image

3. Docker Container(s)

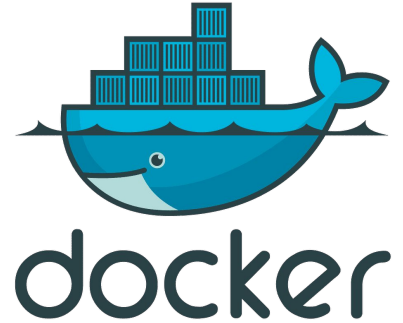


Docker and Python

Step 1 - Install docker!

That should be already installed in your PC!

Otherwise [check this out](#)

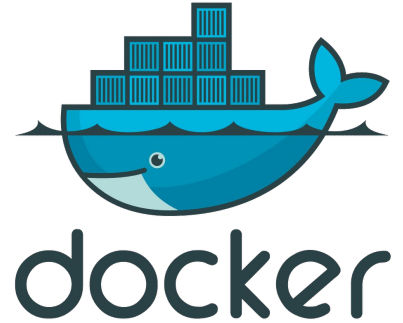
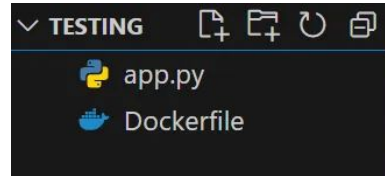


Docker and Python

Step 2 - Create a simple project structure

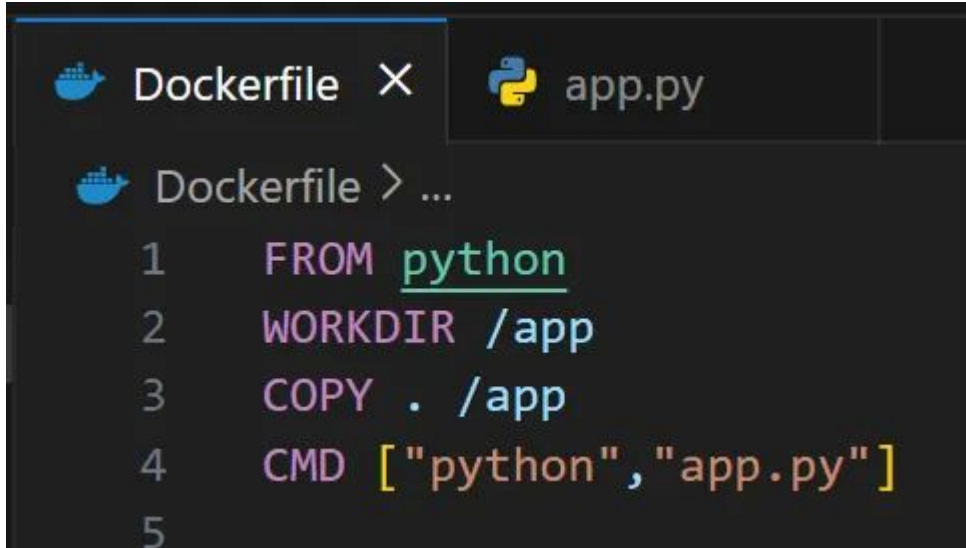
Write inside app.py whatever you want!

Make also a new Dockerfile



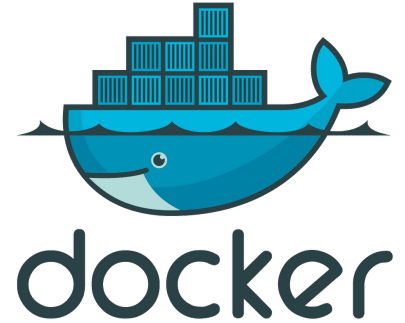
Docker and Python

Step 3 - Configure an appropriate Dockerfile



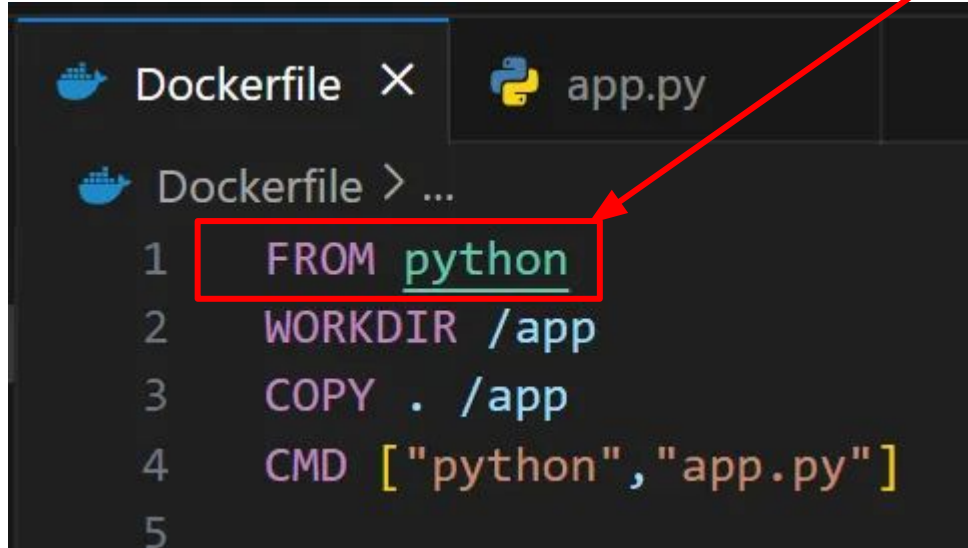
The screenshot shows a code editor with two tabs: 'Dockerfile' and 'app.py'. The 'Dockerfile' tab is active, displaying the following content:

```
Dockerfile > ...  
1 FROM python  
2 WORKDIR /app  
3 COPY . /app  
4 CMD ["python", "app.py"]  
5
```



Docker and Python

Step 3 - Configure an appropriate Dockerfile

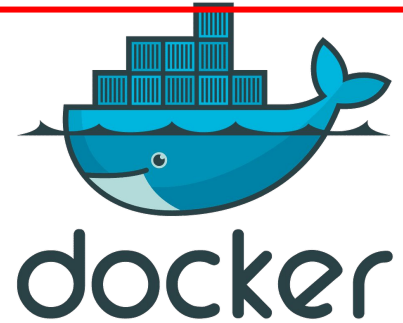


The screenshot shows a code editor with two tabs: 'Dockerfile' and 'app.py'. The 'Dockerfile' tab is active, displaying the following content:

```
1 FROM python
2 WORKDIR /app
3 COPY . /app
4 CMD ["python", "app.py"]
5
```

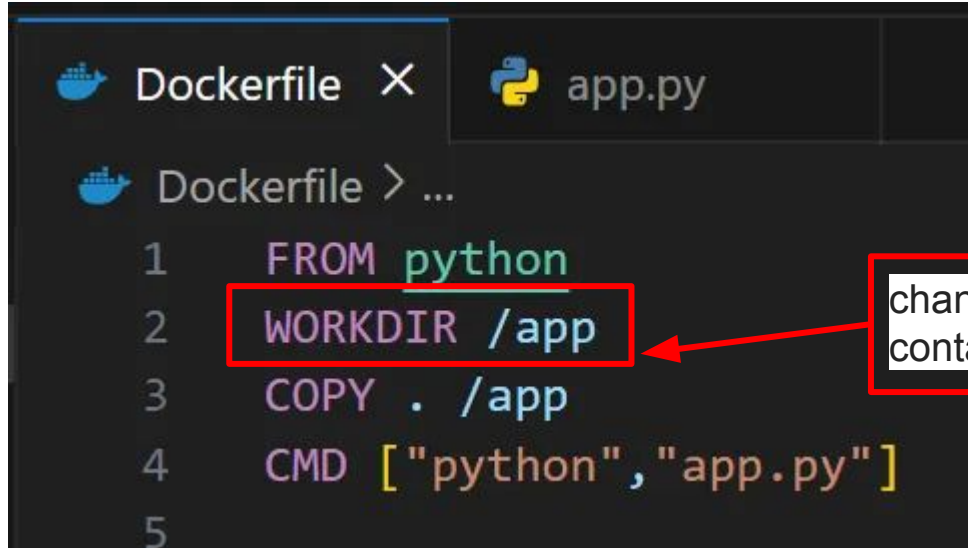
A red box highlights the first line, `FROM python`, and a red arrow points from a text box to this line.

sets the base image as an official Python image



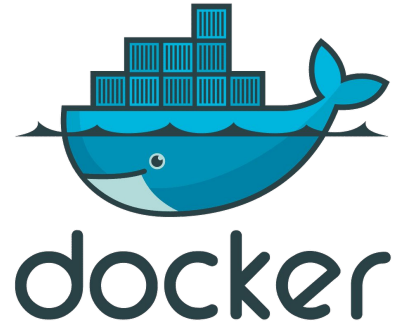
Docker and Python

Step 3 - Configure an appropriate Dockerfile



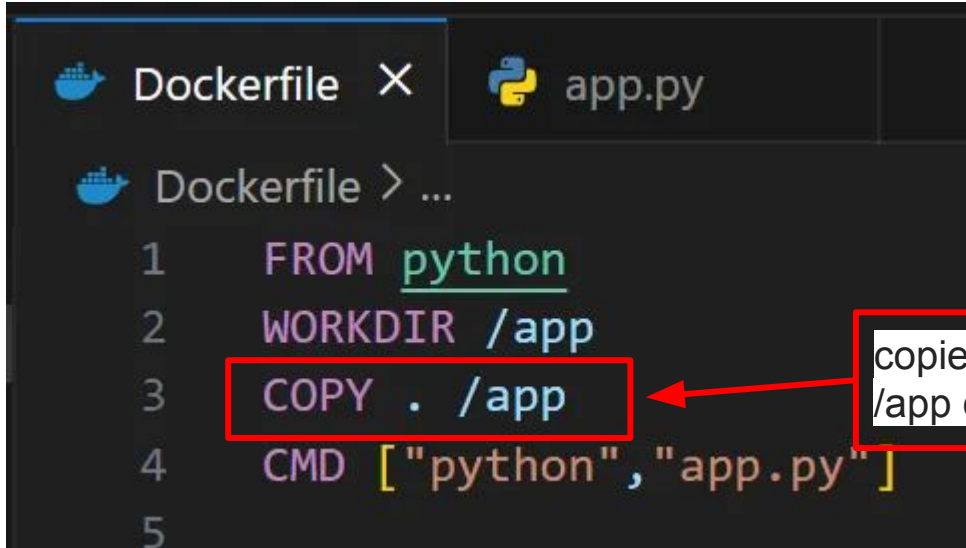
```
Dockerfile ×  app.py
Dockerfile > ...
1  FROM python
2  WORKDIR /app
3  COPY . /app
4  CMD ["python", "app.py"]
5
```

changes the working directory to /app inside the container



Docker and Python

Step 3 - Configure an appropriate Dockerfile

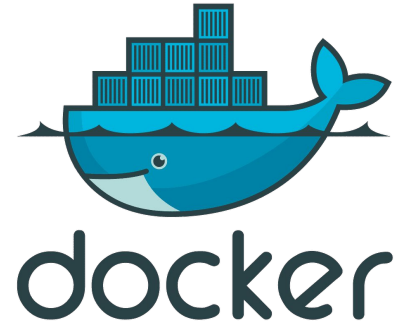


The screenshot shows a code editor with two tabs: 'Dockerfile' and 'app.py'. The 'Dockerfile' tab is active, displaying the following code:

```
1 FROM python
2 WORKDIR /app
3 COPY . /app
4 CMD ["python", "app.py"]
5
```

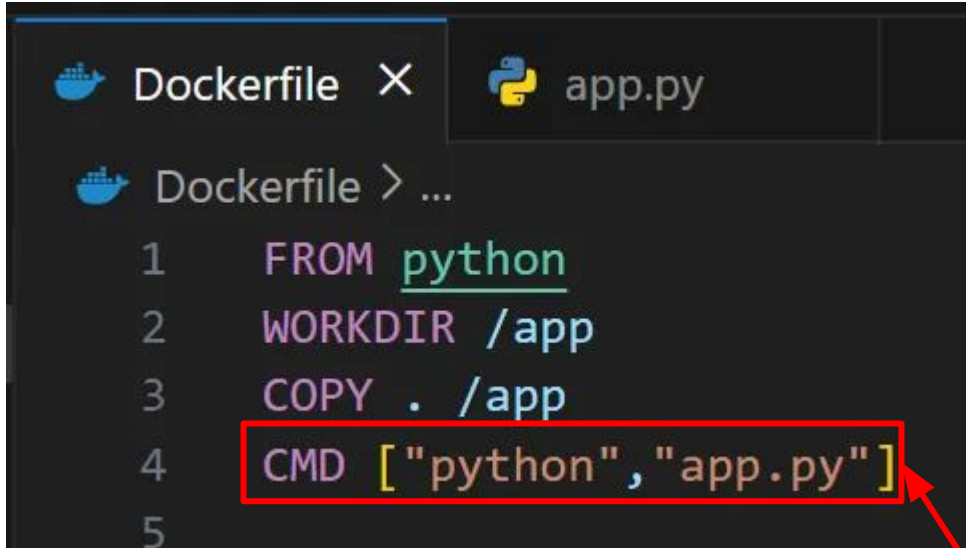
A red box highlights the third line, `COPY . /app`, and a red arrow points from a text box to it.

copies the contents of the current directory into the /app directory within the container.

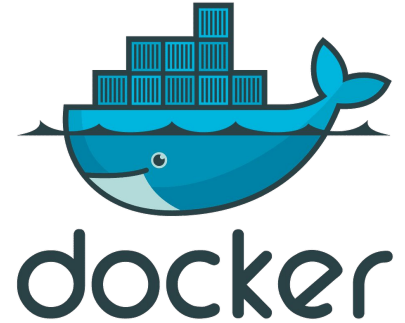


Docker and Python

Step 3 - Configure an appropriate Dockerfile



```
Dockerfile X app.py
Dockerfile > ...
1 FROM python
2 WORKDIR /app
3 COPY . /app
4 CMD ["python", "app.py"]
5
```



specifies that the app.py script should be executed using the Python interpreter when a container is started.

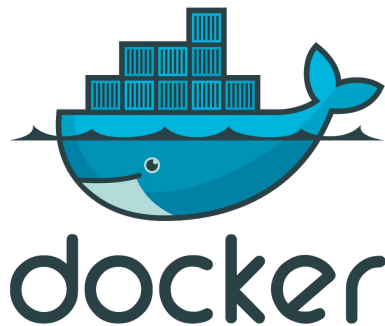
Docker and Python

Step 4 - Build the image from Dockerfile

Use this command `docker build -t mytesting .`

You should get something like this

```
PROBLEMS 11 OUTPUT TERMINAL GITLENS TRUFFLE COMMENTS DEBUG CONSOLE
• PS D:\GITHUB\simple-docker-with-python\testing> docker build -t mytesting .
[+] Building 353.8s (9/9) FINISHED
=> [internal] load .dockerignore 0.1s
=> => transferring context: 2B 0.0s
=> [internal] load build definition from Dockerfile 0.1s
=> => transferring dockerfile: 100B 0.0s
=> [internal] load metadata for docker.io/library/python:latest 9.1s
=> [auth] library/python:pull token for registry-1.docker.io 0.0s
=> [1/3] FROM docker.io/library/python@sha256:02808bfd640d6fd360c30abc4261 343.3s
=> => resolve docker.io/library/python@sha256:02808bfd640d6fd360c30abc4261ad 0.0s
=> => sha256:de4cac68b6165c40cf6f8b30417948c31be03a968e2 49.56MB / 49.56MB 103.9s
=> => sha256:d31b0195ec5f04dfc78eca9d73b5d223fc36a29f54e 24.03MB / 24.03MB 102.5s
=> => sha256:9b1fd34c30b75e7edb20c2fd09a9862697f302ef9ae 64.11MB / 64.11MB 150.9s
0.0s
=> [2/3] WORKDIR /app 1.0s
=> [3/3] COPY . /app 0.1s
=> exporting to image 0.1s
=> => exporting layers 0.1s
=> => writing image sha256:5b87d87d3aa20765864b75d35e11788bc39f1c2d0ad1037d4 0.0s
=> => naming to docker.io/library/mytesting 0.0s
```



Docker and Python

Step 5 - Run the image

Use this command `docker run mytesting`

To run the code

