

12. Writing reproducible code for analysing data

Athina Tzovara

University of Bern

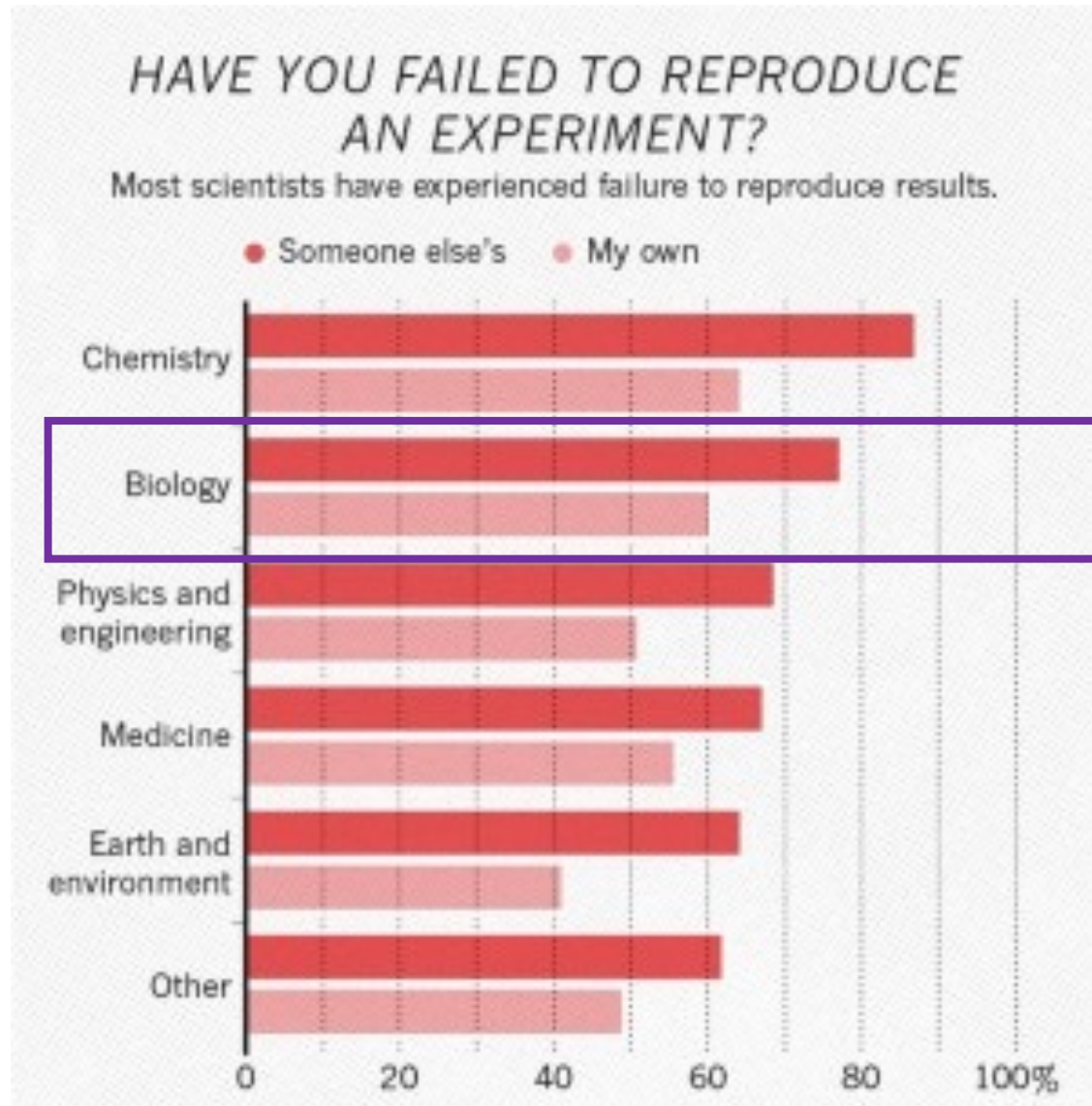


Athina.Tzovara@unibe.ch

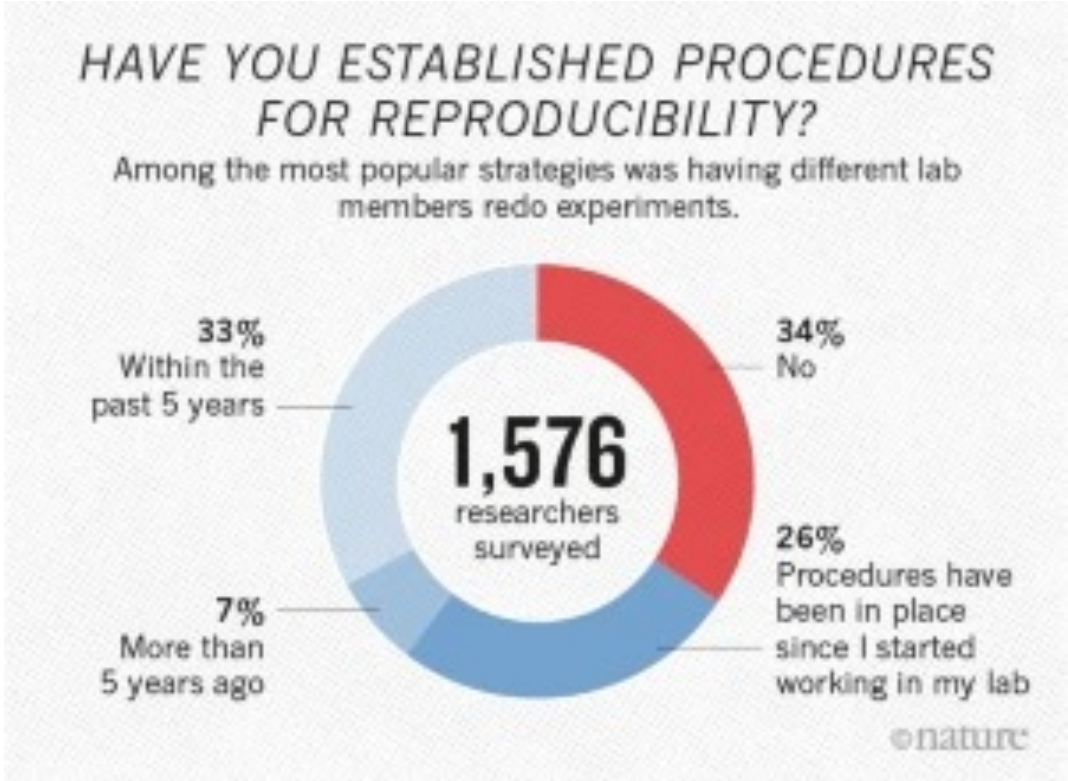
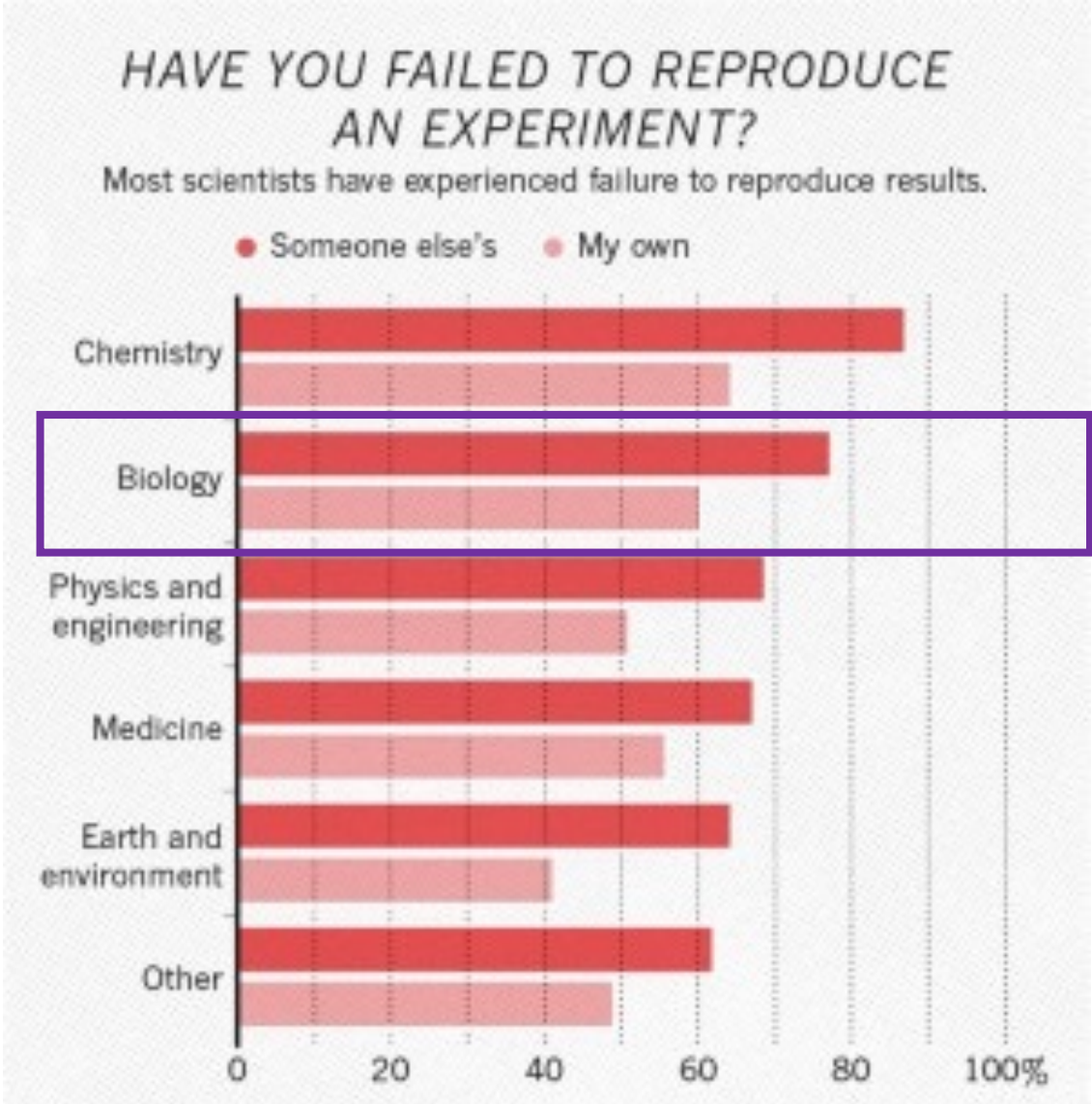
Today

1. Reproducibility in (scientific) programming
2. Virtual environments in python
3. Docker containers
4. Wrapping up & hands-on

Reproducing code and results



Reproducing code and results



Reproducing code and results

1. We may want to use software (libraries) that are not available for our operating system
2. Struggling to install a software due to dependencies
3. Not all dependencies work as desired
4. Everything works on your computer but not on your colleague's computer
5. You need to reproduce a result but you cannot because you used a different version of some software in the past

Reproducing and replicating

		Data	
		Same	Different
		Reproducible	Replicable
Code	Same		
	Different	Robust	Generalizable

An analysis is **not** reproducible if it only runs on your computer!

Why do we care about reproducible coding pipelines for data science?

Ideas?

Why do we care about reproducible coding pipelines for data science?

- **For yourself (in the future)**
 - Re-analyse data; run scripts with different parameters
 - You can document more efficiently how results have been produced
- **For others**
 - Allow others to easily reproduce your findings (Reproducibility crisis)
- **For collaborating**
 - Re-run code and re-use code across projects and colleagues

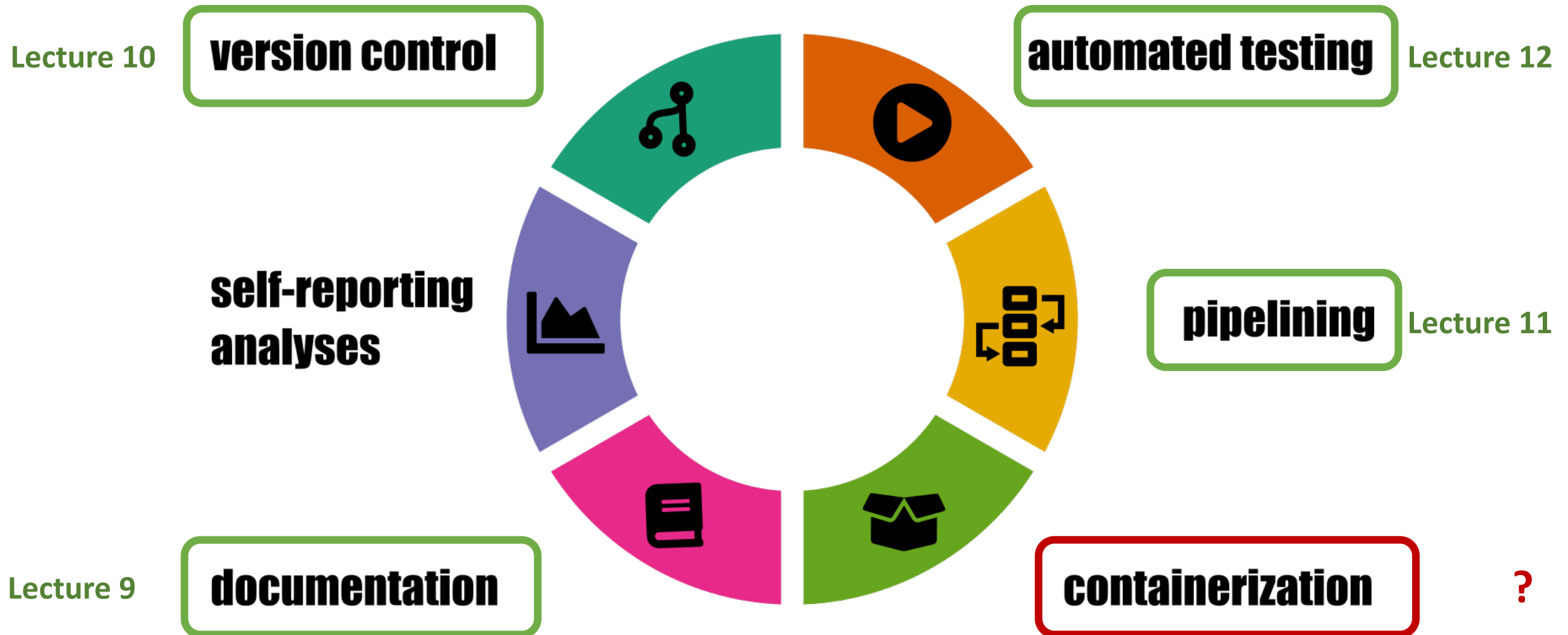
What are the key barriers to reproducibility of code?

Ideas?

What are the key barriers to reproducibility of code?

- A software is no longer available
- Black-box situation: the code is available; no documentation how to run it
- Too many dependencies: documentation is available; the code is available, but too difficult to make it run
- Code rot: code breaks down due to software updates

Hallmarks of good scientific software



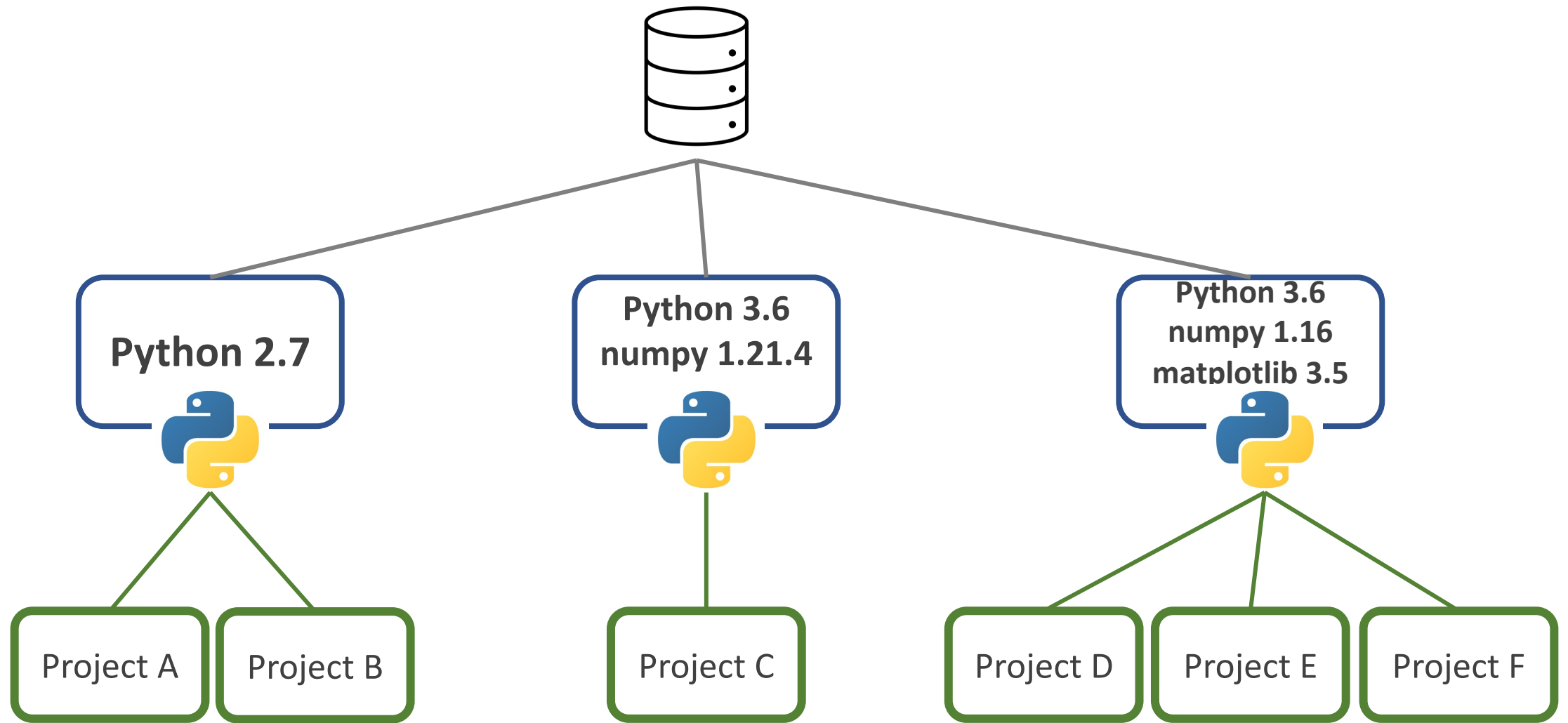
Reproducibility for (Python) coding pipelines

1. Oftentimes data analysis relies on a series of heterogeneous packages, which are brought together via a “pipeline” which links inputs and outputs of various programs
2. We can automate analyses via python / bash scripting, however:
 1. Often no checks for errors
 2. Not easy to partially execute our scripts
 3. Not easy to parallelize

Tools to help with code reproducibility

1. Pyenv
2. Conda
3. Docker

Virtual Environments in python – the idea



Virtual Environments in python – virtualenv

```
$ pip install virtualenv
```

Installing virtualenv

```
$ virtualenv --version
```

Checking the version

Creating a virtual environment inside our folder:

```
$ cd project_folder  
$ virtualenv venv
```

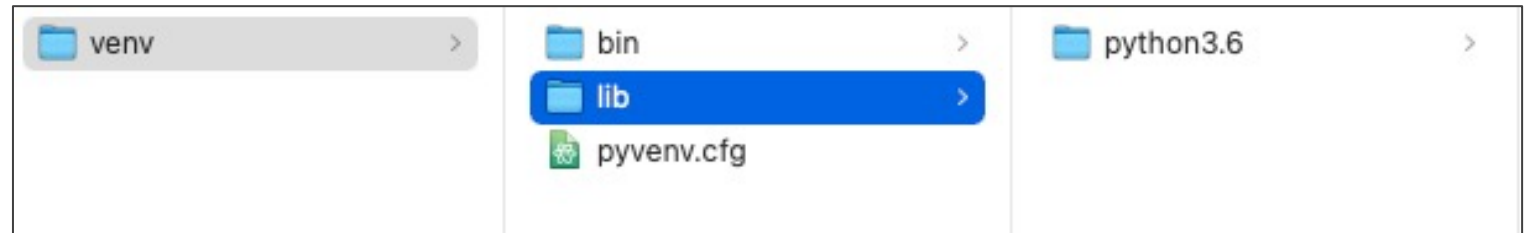
<https://pypi.org/project/virtualenv/>

Virtual Environments in python – virtualenv

```
$ pip install virtualenv
```

```
$ virtualenv --version
```

Result:



Creating a virtual environment inside our folder:

```
$ cd project_folder  
$ virtualenv venv
```

- folder which will contain the Python executable files
- copy of the pip library (use to install other packages)

<https://pypi.org/project/virtualenv/>

Choosing a python interpreter

We choose one interpreter from the virtual environment we just created:

```
$ virtualenv -p /usr/bin/python2.7 venv
```

We can change the interpreter globally with an env variable in ~/.bashrc

```
$ export VIRTUALENVWRAPPER_PYTHON=/usr/bin/python2.7
```

To begin using a virtual environment, it needs to be activated:

```
$ source venv/bin/activate
```

<https://pypi.org/project/virtualenv/>

Activating a virtual environment

To begin using a virtual environment, it needs to be activated:

```
$ source venv/bin/activate
```

The name of the current virtual environment will appear on the left of command line if it is active.

Any package that you install using pip will be placed in the venv folder, isolated from the global Python installation!

Activating a virtual environment

To begin using a virtual environment, it needs to be activated:

```
$ source venv/bin/activate
```

The name of the current virtual environment will appear on the left of command line if it is active.

Any package that you install using pip will be placed in the venv folder, isolated from the global Python installation!

Activation command for windows:

```
C:\Users\SomeUser\project_folder> venv\Scripts\activate
```

Deactivating a virtual environment

To stop using a virtual environment, it needs to be de-activated, simply:

```
$ deactivate
```

The system's default Python interpreter with all its installed libraries is brought back

To delete a virtual environment, just delete its folder. (In this case, it would be `rm -rf venv.`)

“Freezing” a virtual environment

You can “freeze” the current state of the environment packages to keep an environment consistent:

```
$ pip freeze > requirements.txt
```

This will create a requirements.txt file → a list of all the packages in the current environment, and their versions.

You can see the list of installed packages without the requirements format using pip list.

Re-using the same virtual environment

You can install the same packages as in your virtual environment using the same versions:

```
$ pip install -r requirements.txt
```

This can be re-used by any user to re-create your virtual environment!

→ help ensure consistency across installations, deployments, and code contributors.

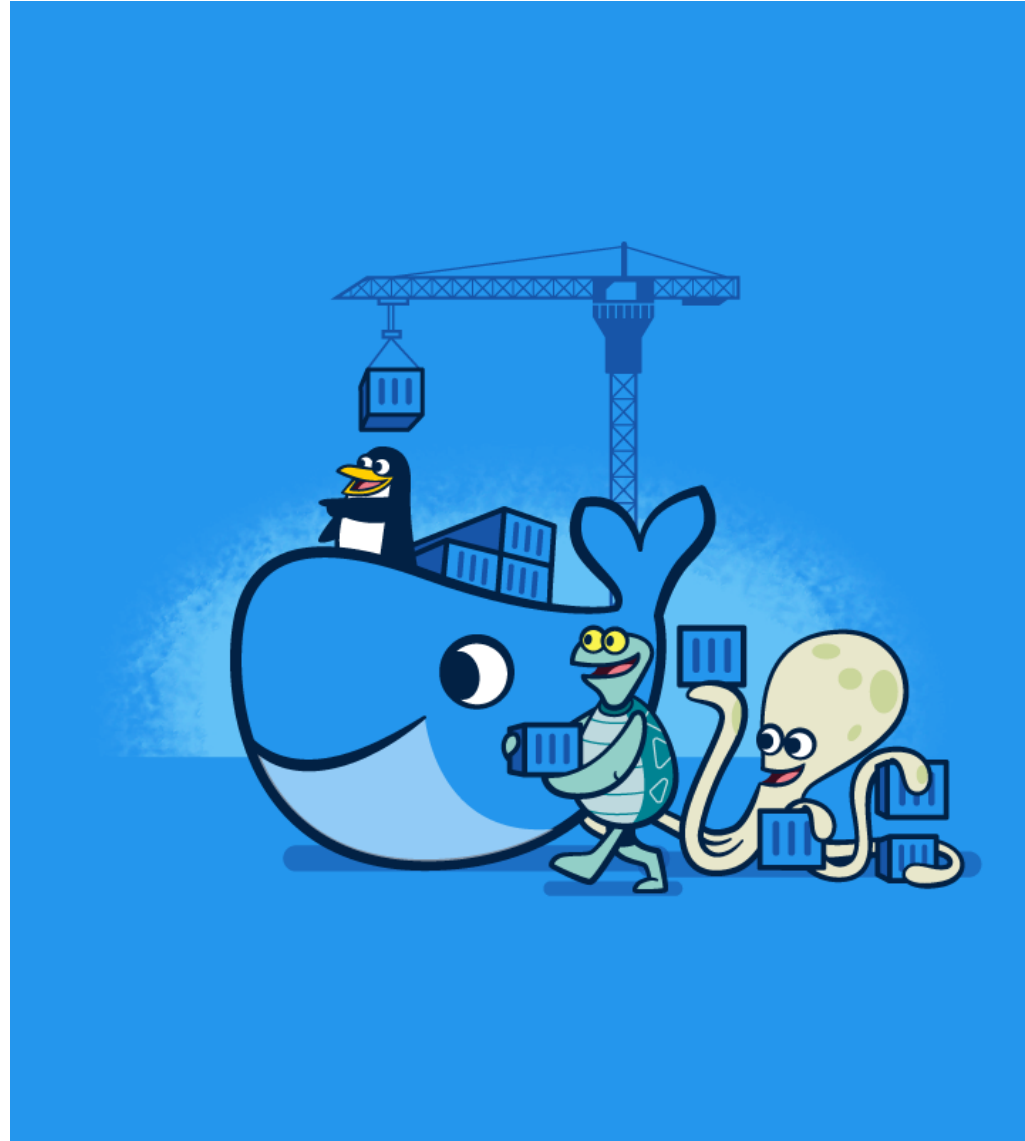
Towards reproducible analyses: Containers

Docker: a tool that lets us build “containers”

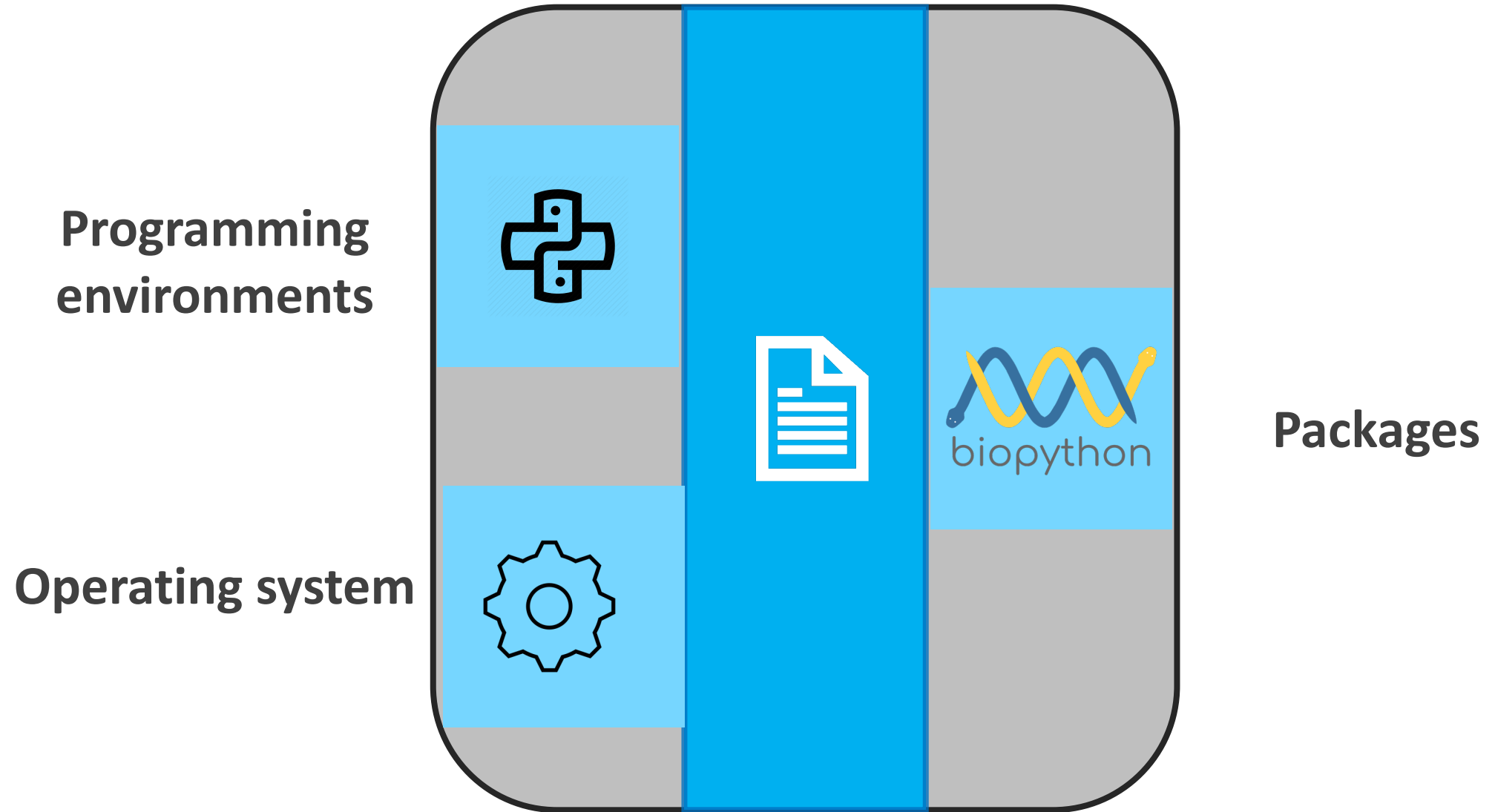
Version control for infrastructure

Easily recreate an entire environment

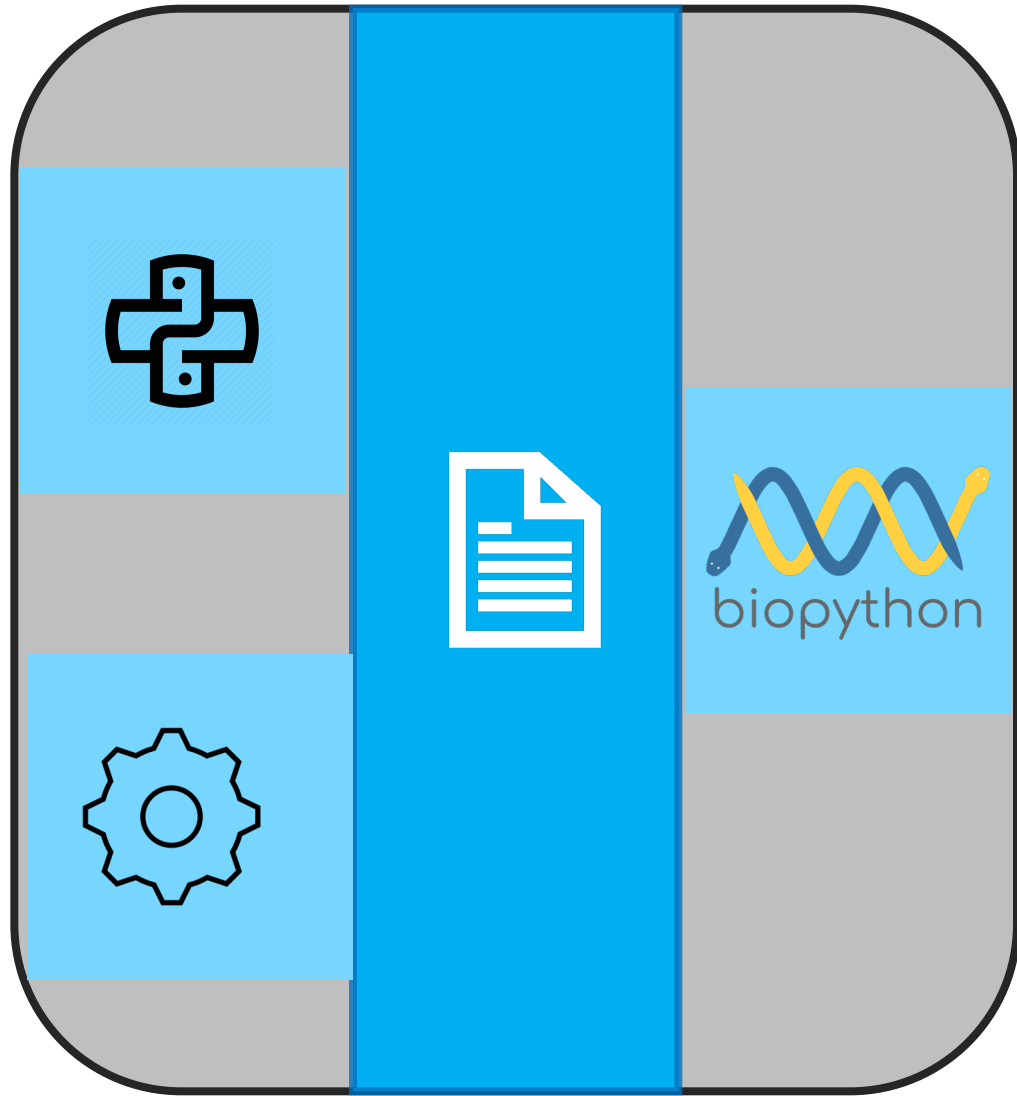
→ Build once, and run everywhere!



What do we mean by containers?



What do we mean by containers?



We achieve:

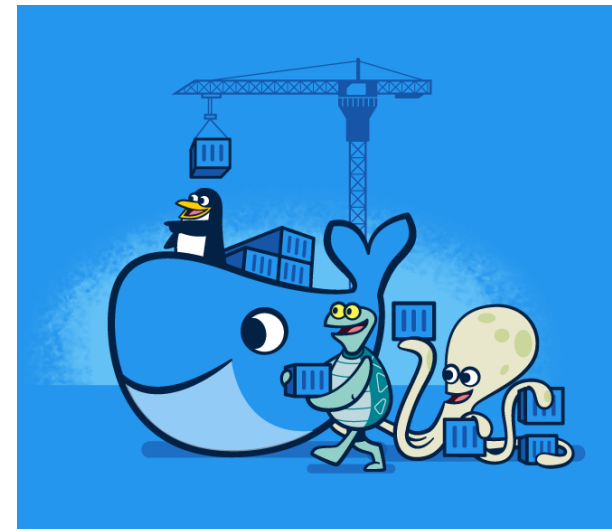
- Standardization
- Portability

We can test our code once and re-use in any environment / Operating system!

Increase reliability and reproducibility

Docker containers

<https://docs.docker.com/>



Important notions:

Dockerfile: a file that contains a set of instructions used to create an **image**

Image: used to build and save “snapshots” of an environment

Container: an instantiated, live **image** that runs a collection of processes.

Towards reproducible code: Overview of containers

<https://docs.docker.com/>

Once your Docker application is running → check that Docker is installed and the command line tools are working correctly:

Bash

```
$ docker container ls
```

Output

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
--------------	-------	---------	---------	--------	-------	-------

Docker images

We can now run containers from named Docker images:
`docker container run`

Bash

```
$ docker container run hello-world
```

Output

Hello from Docker!

This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
(amd64)
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

```
$ docker run -it ubuntu bash
```

Share images, automate workflows, and more with a free Docker ID:

<https://hub.docker.com/>

For more examples and ideas, visit:

<https://docs.docker.com/get-started/>

Running containers

We can now run containers from named Docker images:

`docker container run`

1. Starting a running container
>> “alive” or “inflated” version of the container
2. Performing default actions
>> if the container has one
3. Shutting down the container
>> The container exits. The image is still there but nothing is actively running

Running containers

We will now try out the "alpine" container:

```
[staff-207-140:VirtualEnv athina$ docker container run alpine
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
9b3977197b4f: Pull complete
Digest: sha256:21a3deaa0d32a8057914f36584b5288d2e5ecc984380bc0118285c70fa8c9300
Status: Downloaded newer image for alpine:latest
[staff-207-140:VirtualEnv athina$ docker container run alpine
[staff-207-140:VirtualEnv athina$ docker container run alpine cat etc/os-release
NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.15.0
PRETTY_NAME="Alpine Linux v3.15"
HOME_URL="https://alpinelinux.org/"
BUG_REPORT_URL="https://bugs.alpinelinux.org/"
```

We can run a
command on the
docker

Running containers

We will now try out the "alpine" container:

```
[staff-207-140:VirtualEnv athina$ docker container run alpine
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
9b3977197b4f: Pull complete
Digest: sha256:21a3deaa0d32a8057914f36584b5288d2e5ecc984380bc0118285c70fa8c9300
Status: Downloaded newer image for alpine:latest
[staff-207-140:VirtualEnv athina$ docker container run alpine
[staff-207-140:VirtualEnv athina$ docker container run alpine cat etc/os-release
NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.15.0
PRETTY_NAME="Alpine Linux v3.15"
HOME_URL="https://alpinelinux.org/"
BUG_REPORT_URL="https://bugs.alpinelinux.org/"
```

We can run a
command on the
docker

```
[staff-207-140:VirtualEnv athina$ docker container run alpine echo "Hello World"
Hello World
```

Running containers interactively

Previously, Docker has started the container, run a command, and then immediately shut down the container.

Can we keep the container running so we can log into it and test more commands?

→ Use interactive flag `-i t` and provide a shell (`sh` etc.) as our command:

```
staff-207-140:VirtualEnv athina$ docker container run -it alpine sh

/ #
/ # pwd
/
/ # ls
bin      etc      lib      mnt      proc     run      srv      tmp      var
dev      home     media    opt      root     sbin     sys      usr
/ # whoami
root
/ #
```


Bringing python inside containers

Build a container image: we can start by installing software inside our container

1. We start the interactive alpine container:

Bash

```
$ docker container run -it alpine sh
```

2. We can check whether python3 is already installed:

Bash

```
/# python3
```

Output

```
sh: python3: not found
```

Bringing python inside docker

Build a container image: we can start by installing software inside our container

3. We install python 3 in our container:

Bash

```
/# apk add --update python3 py3-pip python3-dev
```

4. We can then check whether our installation has worked:

Bash

```
/# python3 --version
```

5. We can add packages:

Bash

```
/# pip install cython
```

Bringing python inside docker via a Dockerfile

Dockerfile: a plain text file with keywords and commands that can be used to create a new container image:

Output

```
FROM <EXISTING IMAGE>  
RUN <INSTALL CMDS FROM SHELL>  
RUN <INSTALL CMDS FROM SHELL>  
CMD <CMD TO RUN BY DEFAULT>
```

General layout

Bringing python inside docker via a Dockerfile

Dockerfile: a plain text file with keywords and commands that can be used to create a new container image:

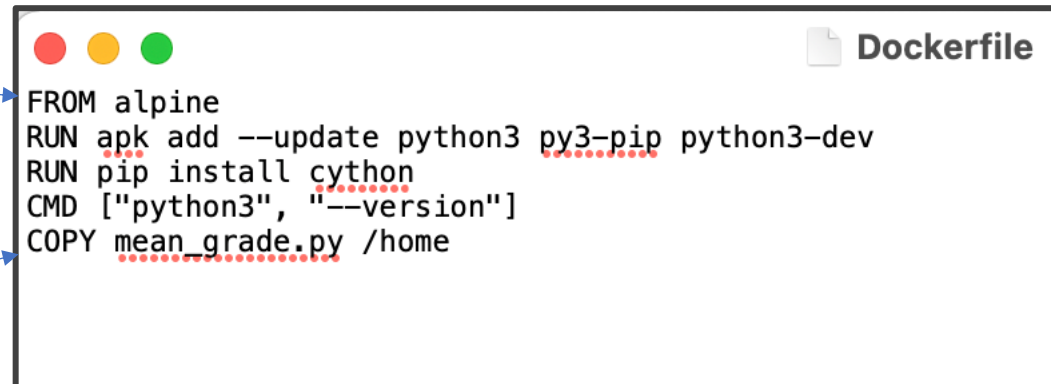
Output

```
FROM <EXISTING IMAGE>  
RUN <INSTALL CMDS FROM SHELL>  
RUN <INSTALL CMDS FROM SHELL>  
CMD <CMD TO RUN BY DEFAULT>
```

General layout

Exemplar **Dockerfile**

Existing image
What we want to install
Commands we want to run
Local files we want to include




```
FROM alpine  
RUN apk add --update python3 py3-pip python3-dev  
RUN pip install cython  
CMD ["python3", "--version"]  
COPY mean_grade.py /home
```

Building a new Docker image

We can now build a container image:

```
$ docker image build -t USERNAME/CONTAINERNAME .
```

 Your username on dockerhub

 The name of your container

```
MacBook-Air-2:Docke athina$ docker image build -t advancedpython/alpine-python .
[+] Building 5.7s (7/7) FINISHED
=> [internal] load build definition from Dockerfile                                0.0s
=> => transferring dockerfile: 155B                                              0.0s
=> [internal] load .dockerignore                                                0.0s
=> => transferring context: 2B                                                  0.0s
=> [internal] load metadata for docker.io/library/alpine:latest                0.0s
=> [1/3] FROM docker.io/library/alpine                                         0.0s
=> [2/3] RUN apk add --update python3 py3-pip python3-dev                     3.7s
=> [3/3] RUN pip install cython                                               1.5s
=> exporting to image                                                         0.4s
=> => exporting layers                                                         0.4s
=> => writing image sha256:e5ae1d155df0174027c3472c5d5d4701b8800374bccd7      0.0s
=> => naming to docker.io/advancedpython/alpine-python                       0.0s
```

Running python scripts from outside the container

If we try to run local scripts through our container, they will fail:

Bash

```
$ docker container run alice/alpine-python python3 sum.py
```

Output

```
python3: can't open file 'sum.py': [Errno 2] No such file or directory
```

The container and its filesystem is separate from our host computer's filesystem

When the container runs, it can't see anything outside itself (e.g. any files on our computer).

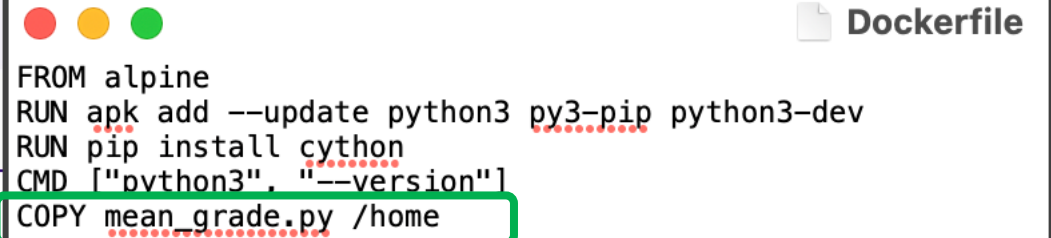
We need to create a link between the directory on our computer and the container.

Running python scripts from outside the container

The simplest way to use scripts from our computer inside a container:
In the **Dockerfile**, add:

Code

```
COPY sum.py /home
```



```
FROM alpine
RUN apk add --update python3 py3-pip python3-dev
RUN pip install cython
CMD ["python3", "--version"]
COPY mean_grade.py /home
```

And build again a docker image:

Bash

```
$ docker image build -t alice/alpine-sum .
```

Running python scripts from outside the container

We can now run again our container :

Bash

```
$ docker container run -it alice/alpine-sum sh
```

And once inside, run our script through python:

Bash

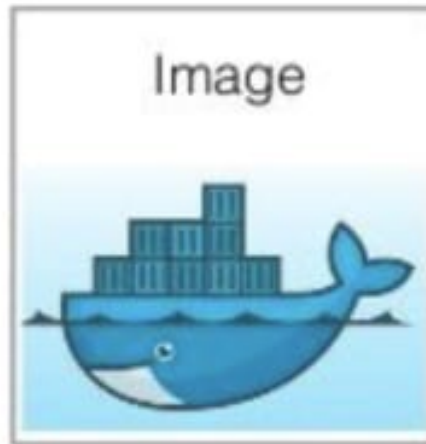
```
/# python3 /home/sum.py
```


Docker containers and images: Summary

```
1 FROM ubuntu:16.04
2 RUN apt-get update && apt-get install -y python3 python3-dev python3-pip
3
4 WORKDIR /app
5 COPY . /app
6 RUN pip3 install -r requirements.txt
7
8 EXPOSE 80
9
10 CMD ["python3", "app.py"]
```

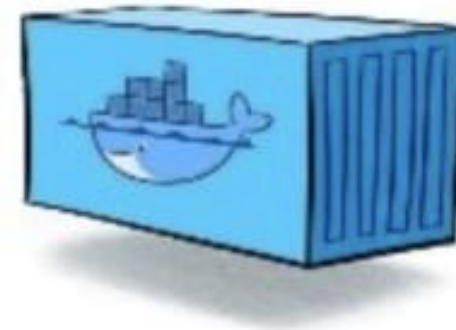
Dockerfile

build



Docker Image

run



Docker Container

Hands-on: accessing and running a docker container

- Install Docker on your computer

- Download a docker image for lecture 13:

```
docker run -d -p 80:80 advancedpython/lecture13
```

- Run the docker container:

```
docker container run -it  
advancedpython/lecture13 sh
```

- Run a script inside the container:

```
python3 /home/Lecture13_Example.py
```

Reproducible code: Summary

- **Reproducible code**: scientific programming; rendering analyses more robust by increasing reproducibility
- In python: **Virtual environments** (`virtualenv`):
 - Help us create 'virtual environments' where we can have python installations and packages; multiple python versions in parallel
- **Containers** (not just for python):
 - A virtual operating system
 - Contain and OS, programming environments, software
 - Useful for testing new code

Today

1. Reproducibility in (scientific) programming
2. Virtual environments in python
3. Docker containers
4. Wrapping up & hands-on