The goal of this exercise is to explore and learn how to use Docker containers.

1 Getting started: Install Docker

Go to https://docs.docker.com/install/ and download the latest Docker Desktop version available for your operating system.

After the install make sure docker has been correctly installed and configured for your environment by opening a terminal and typing: `docker version`

The output should be something like the following:

```
Client:
 Cloud integration: v1.0.22
 Version:           20.10.12
 API version:       1.41
 Go version:        go1.16.12
 Git commit:        e91ed57
 Built:             Mon Dec 13 11:46:56 2021
 OS/Arch:           darwin/amd64
 Context:           default
 Experimental:      true

Server: Docker Desktop 4.5.0 (74594)
 Engine:
  Version:          20.10.12
  API version:      1.41 (minimum version 1.12)
  Go version:       go1.16.12
  Git commit:       459d0df
  Built:            Mon Dec 13 11:43:56 2021
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          1.4.12
  GitCommit:        7b11cfaabd73bb80907dd23182b9347b4245eb5d
 runc:
  Version:          1.0.2
  GitCommit:        v1.0.2-0-g52b36a2
 docker-init:
  Version:          0.19.0
  GitCommit:        de40ad0
```

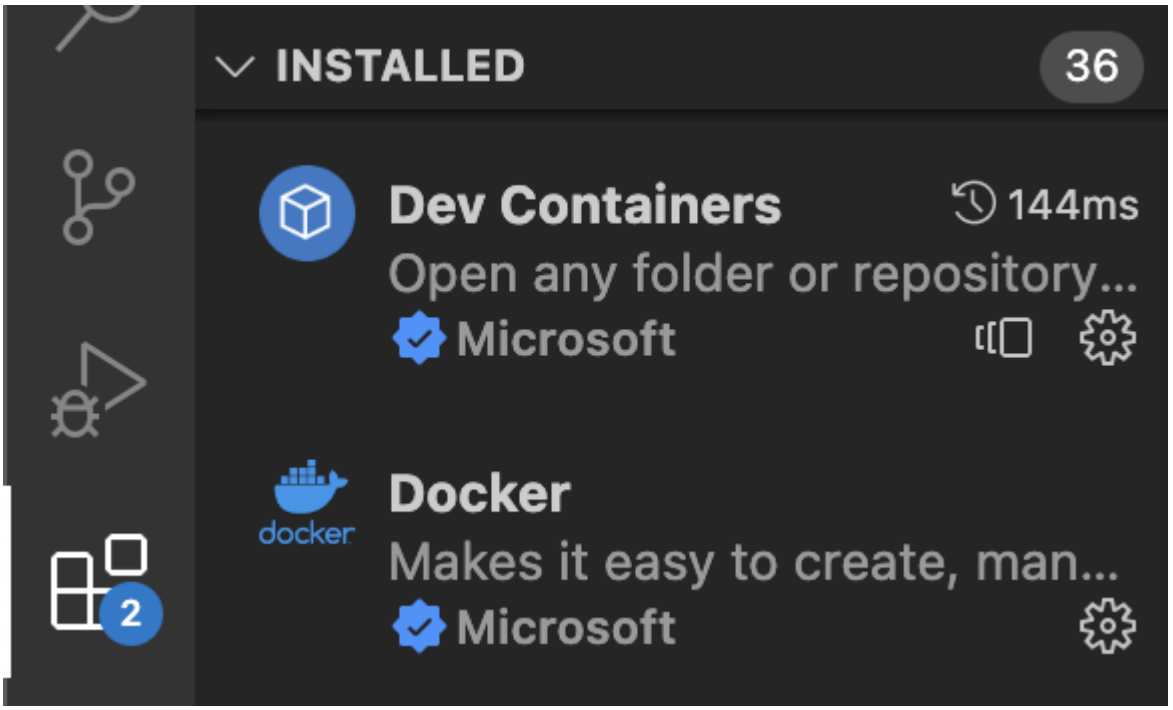
2 Install VS Code

If not already done, download and install Visual Studio Code for your operating system:

https://code.visualstudio.com/download

3 Install Dev Container

1. Create a new project folder, for example `$HOME/courses/pacp/lab/01-devcontainer/`
2. Open the new folder inside VSCode.
3. install the Dev Container extension



4 Run Dev Container

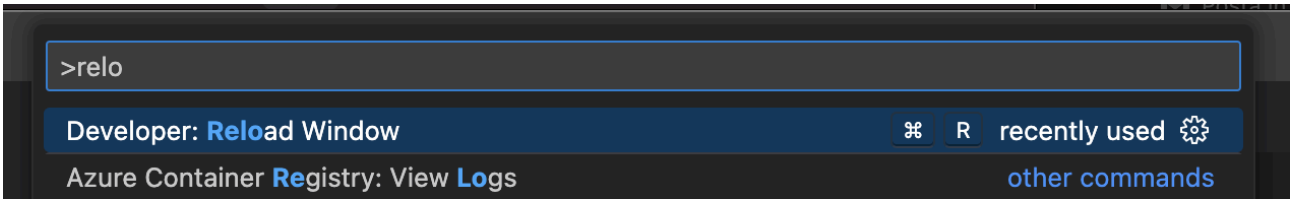
Download the `devcontainer_setup.zip` file from the page of the course.

Unzip the file and copy the resulting directory into your project folder.

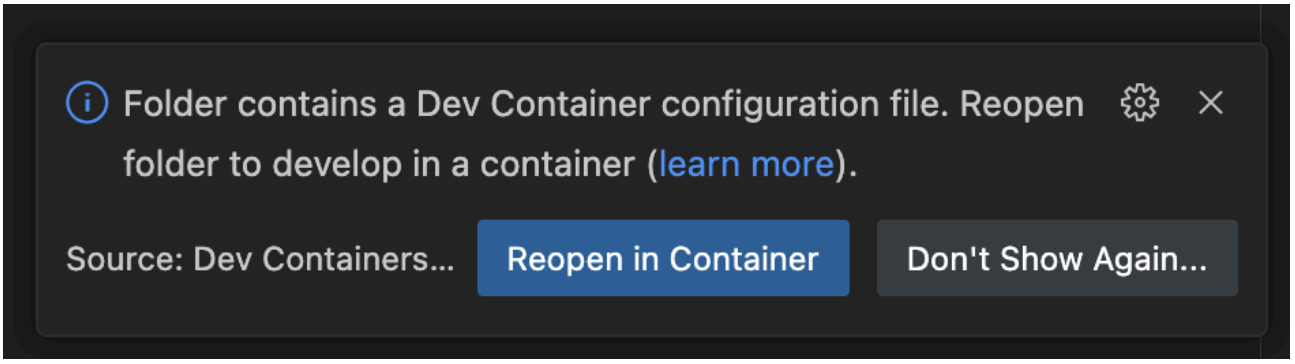
IMPORTANT! Copy the entire directory in the root folder of your project and rename it as `.devcontainer` (with a dot in front of the name). **Otherwise, it will not work.**

At this point, you need to restart VSCode for the changes to take effect.

Press F1 and search for the “Reload Window” command:

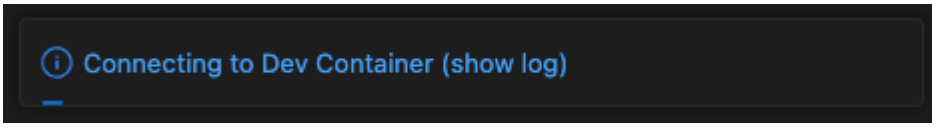


When VSCode restarts it will ask you to reopen the folder inside a container.
Agree and press “Reopen in Container”:



NOTE. If you don't see the message, either you missed renaming the `devcontainer` to `.devcontainer`, or you missed putting the `.devcontainer` folder into the **root folder** of the project.

The command will build Python 3.13 from sources and it will take a while. You can press the “show log) button to have feedback about what is going on:



To verify the container is running, open a new terminal inside VSCode. You should see a prompt similar to the following:

```
root@8b13d0dd3a3d:/workspaces/01-devcontainer#
```

If you run `python --version` you should get a version higher than 3.13 (e.g. 3.14.0a3+).

NOTE. The command `python` will run the version with the GIL disabled. The command `python_gil` will run the version with the GIL enabled. Try both commands to see the differences.

5 Play around with threads

Copy the `gil_test.py` file from the page of the course.

The program will run the same code in three different modes: sequentially, using multiple threads, and using multiple processes. We will analyse the differences between those different modes during the course.

Run the program with both `python` and `python_gil` commands and check the output.

Copy the `multitread_test.py` file from the page of the course.

The program takes as an optional argument the number of threads to run.
If no arguments are provided, it will run as many threads as the number of available cores.

NOTE. It is not the number of cores of your CPU but **the number of cores of the container**. For example, my laptop has 8 cores, but the container can access only 4 cores.

The program will run the same CPU intensive computation for each thread. It will count the 30th step of the Fibonacci sequence. On my laptop, a single execution takes about 1.3 seconds.

Single thread

We start running a single threaded execution:

```
python multithread_test.py 1
```

The outcome of the program should be the following:

```
GIL=False
Thread[0] => fib(30) = 1346269 (took 1.24 seconds)
--- The overall time is 1.25 seconds ---
```

It tells that the GIL is disabled and shows the time taken by each thread to run (in this case 1) and the time taken by the overall program to execute.

One thread per core

We run the program again without passing any argument:

```
python multithread_test.py
```

You should see an outcome similar to the following:

```
GIL=False
```

```
Thread[0] => fib(30) = 1346269 (took 1.4 seconds)
Thread[1] => fib(30) = 1346269 (took 1.43 seconds)
Thread[3] => fib(30) = 1346269 (took 1.44 seconds)
Thread[2] => fib(30) = 1346269 (took 1.47 seconds)
--- The overall time is 1.48 seconds ---
```

In my case, the container is enabled to access 4 cores. The program runs the Fibonacci sequence in parallel on each core.

Every thread takes more or less the same time, and the **overall computation** takes as much time as the **slowest thread**.

Two threads per core

Now we will run the same computation forcing as many threads as twice the number of cores.
In my case, the container has 4 cores, so I ran the program with 8 threads:

```
python multithread_test.py 8
```

The outcome of the program was the following:

```
GIL=False
Thread[6] => fib(30) = 1346269 (took 2.32 seconds)
Thread[5] => fib(30) = 1346269 (took 2.65 seconds)
Thread[2] => fib(30) = 1346269 (took 2.73 seconds)
Thread[1] => fib(30) = 1346269 (took 2.84 seconds)
Thread[0] => fib(30) = 1346269 (took 2.86 seconds)
Thread[4] => fib(30) = 1346269 (took 2.86 seconds)
Thread[3] => fib(30) = 1346269 (took 2.88 seconds)
Thread[7] => fib(30) = 1346269 (took 2.94 seconds)
--- The overall time is 3.01 seconds ---
```

As expected, the overall time is almost twice the previous execution. This is because only 4 threads can run in parallel while the execution of the other 4 is interleaved.

As you can notice, even the execution time of every thread is slower. This is because access to the CPU cores is evenly distributed among threads. Since there are more threads than cores, every thread cannot access the CPU full-time. It must wait for other threads to execute, resulting in a slower average execution.