

# Bioinformatics session

3rd Annual workshop on  
bioinformatics and variant  
interpretation in InPreD

[https://inpred.github.io/25-06\\_bioinfo\\_ws/bioinfo\\_ws](https://inpred.github.io/25-06_bioinfo_ws/bioinfo_ws)



# **1. Unit testing**

## What is unit testing?

- test smallest piece of code that can be logically isolated in software application (function, subroutine, method)
- the smaller the better - more granular view of what is going on; also faster
- should not cross systems (database, filesystem, network) -> integration and functional tests

## Example

```
# calculator.py
def add(x, y):
    """add numbers"""
    return x + y
```

```
# test_calculator.py
import calculator

def test_add():
    assert calculator.add(1, 2) == 3
```

## **Why do we need unit testing?**

- early defect detection
- code quality improvement
- facilitates refactoring
- faster development cycles
- better documentation
- enables more frequent releases

## How to design a unit test?

- identify the unit (function, method)
- what is its functionality?
- what is the input (correct and incorrect)?
- how to handle incorrect input? (edge cases, invalid data)
- what does it return?
- positive and negative results should be tested

## Set up unit testing for your functions

- install pytest

```
$ pip install pytest
```

- add your function to a module at `my_module/my_module.py`
- add your unit test at `my_module/tests/my_module_test.py`
- in the test file import your module `from my_module.my_module import my_function`

# First exercise

- go to [https://github.com/InPreD/25-06\\_bioinfo\\_ws\\_unit\\_testing](https://github.com/InPreD/25-06_bioinfo_ws_unit_testing)

The screenshot shows the GitHub interface for the repository **25-06\_bioinfo\_ws\_unit\_testing** (Public). At the top, there are buttons for **Edit Pins**, **Watch** (1), **Fork** (0), and **Star** (0). Below the repository name, it shows **main** branch, **1 Branch**, and **0 Tags**. A search bar labeled "Go to file" and an **Add file** button are present. A green **<> Code** button is highlighted. A dropdown menu is open from the **Code** button, showing two tabs: **Local** and **Codespaces**. The **Codespaces** tab is active, displaying a list of workspaces. The first workspace is **shiny space eureka** on the **main** branch, with a status of **No changes** and a last update time of **3d**. A tooltip **Create a codespace on main** is visible over the **main** branch link. Below the workspace list, it states: **Codespace usage for this repository is paid for by marrip.**

The repository's **About** section on the right indicates it is a **codespace template for unit testing workshop**. It includes links to the **Readme**, **AGPL-3.0 license**, **Activity**, **Custom properties**, **0 stars**, **1 watching**, and **0 forks**. There are also links to **Report repository**, **Releases** (No releases published, [Create a new release](#)), **Packages** (No packages published, [Publish your first package](#)), and **Languages** (A bar chart shows **Dockerfile** at **100.0%**). The **Suggested workflows** section is also visible, based on the tech stack.



# First exercise

← →

25-06\_bioinfo\_ws\_unit\_testing [Codespaces: shiny space eureka]

👤

☰

EXPLORER

⋮

25-06\_BIOINFO\_WS\_UNIT\_TESTING [CODESPA...]

> .devcontainer

🔑 LICENSE

📄 README.md

🔍

🔗

🔧

📁

🐙

👤

⚙️

> GLIEDERUNG

> ZEITACHSE

[Vorschau] README.md ×

## 25-06\_bioinfo\_ws\_unit\_testing

codespace template for unit testing workshop

PROBLEME

AUSGABE

DEBUGGING-KONSOLE

TERMINAL

PORTS

🐚 bash + ▾ 📄 🗑️ ⋮ ^ ×

root@codespaces-defab7:/workspaces/25-06\_bioinfo\_ws\_unit\_testing#

< Codespaces: shiny space eureka

🔗 main ↻

⊗ 0 ⚠️ 0 🗨️ 0

👤 Layout: German 🔔

## First exercise

- pytest was already installed in the codespace
- the suggested layout was already applied
- create a branch for your work:

```
$ git checkout -b unit-tests-<your name>
```

- start with the first exercise in `first/tests/first_test.py`
- whenever you are done, commit your changes (use [commit message conventions](#)):

```
$ git add first/tests/first_test.py  
$ git commit -m "test: <your commit message>"
```

- and we push them to GitHub:

```
$ git push --set-upstream origin unit-tests-<your name>
```

## Handle exceptions in unit tests

- functions can raise exceptions and we would like to test for those
- import `pytest` to have access to `raises()`
- add `with` -block to handle the exception:

```
import calculator
import pytest

def test_add_exception():
    with pytest.raises(TypeError):
        assert add("one", "two") == None
```

## Second exercise

- continue with the second exercise in `second/tests/second_test.py`
- whenever you are done, commit your changes (use [commit message conventions](#)):

```
$ git add second/tests/second_test.py  
$ git commit -m "test: <your commit message>"
```

- and we push them to GitHub:

```
$ git push
```

## Make unit tests table-driven by using parametrize

- having more than one test case results in repeating a lot of code (one function per test case)
- to condense this as much as possible (ideally one unit test per function), we can use the `pytest` decorator `parametrize`
- again, import `pytest` to gain access to the decorator
- add the decorator `@pytest.mark.parametrize` as a header to your function
- define the required variables (input, exception, output)
- add your test cases as a list of tuples (one tuple per case)
- also use `nullcontext` from the module `contextlib` to account for cases without exceptions

```
import calculator
import pytest

from contextlib import nullcontext

@pytest.mark.parametrize(
    "x, y, exception, want",
    [
        (1, 2, nullcontext(), 3),
        ("one", "two", pytest.raises(TypeError), None)
    ]
)
def test_add(x, y, exception, want):
    with exception:
        assert add(x, y) == want
```

## Third exercise

- continue with the third exercise in `third/tests/third_test.py`
- whenever you are done, commit your changes (use [commit message conventions](#)):

```
$ git add third/tests/third_test.py  
$ git commit -m "test: <your commit message>"
```

- and we push them to GitHub:

```
$ git push
```

# Use GitHub action to automatically run tests on push

- add `.github/workflows/main.yml` :

```
name: Python test
on: push

jobs:
  test:
    name: Run unit tests
    runs-on: ubuntu-latest
    steps:
      -
        name: Check out the repo
        uses: actions/checkout@v4
      -
        name: Set up Python 3.12.8
        uses: actions/setup-python@v4
        with:
          python-version: 3.12.8
      -
        name: Install dependencies
        run: pip install -r requirements.txt
      -
        name: Unit testing
        uses: pavelzw/pytest-action@v2
        with:
          verbose: true
          emoji: true
          job-summary: true
          custom-arguments: -q
          click-to-expand: true
          report-title: 'Bioinfo workshop unit testing'
```



## Use GitHub action to automatically run tests on push

- if you don't want to write all of that, merge the branch containing the file into your branch:

```
$ git merge add-github-action
```

## Fourth exercise

- write unit tests for the functions in `fourth/fourth.py`
- whenever you are done, commit your changes (use [commit message conventions](#)):

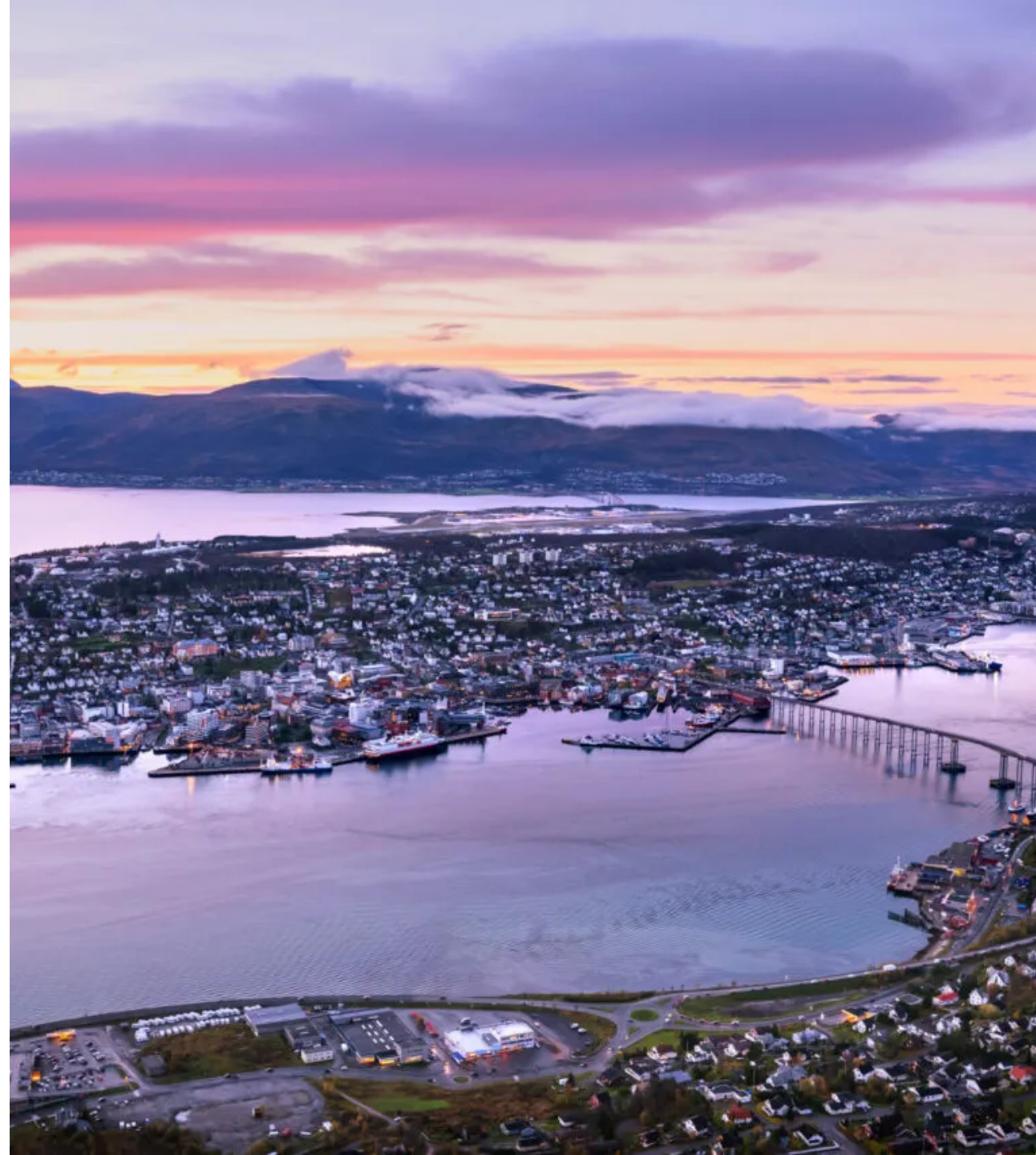
```
$ git add third/tests/third_test.py  
$ git commit -m "test: <your commit message>"
```

- and we push them to GitHub:

```
$ git push
```

Thank you for your attention!

Day 1 done!



## **2. Nextflow**

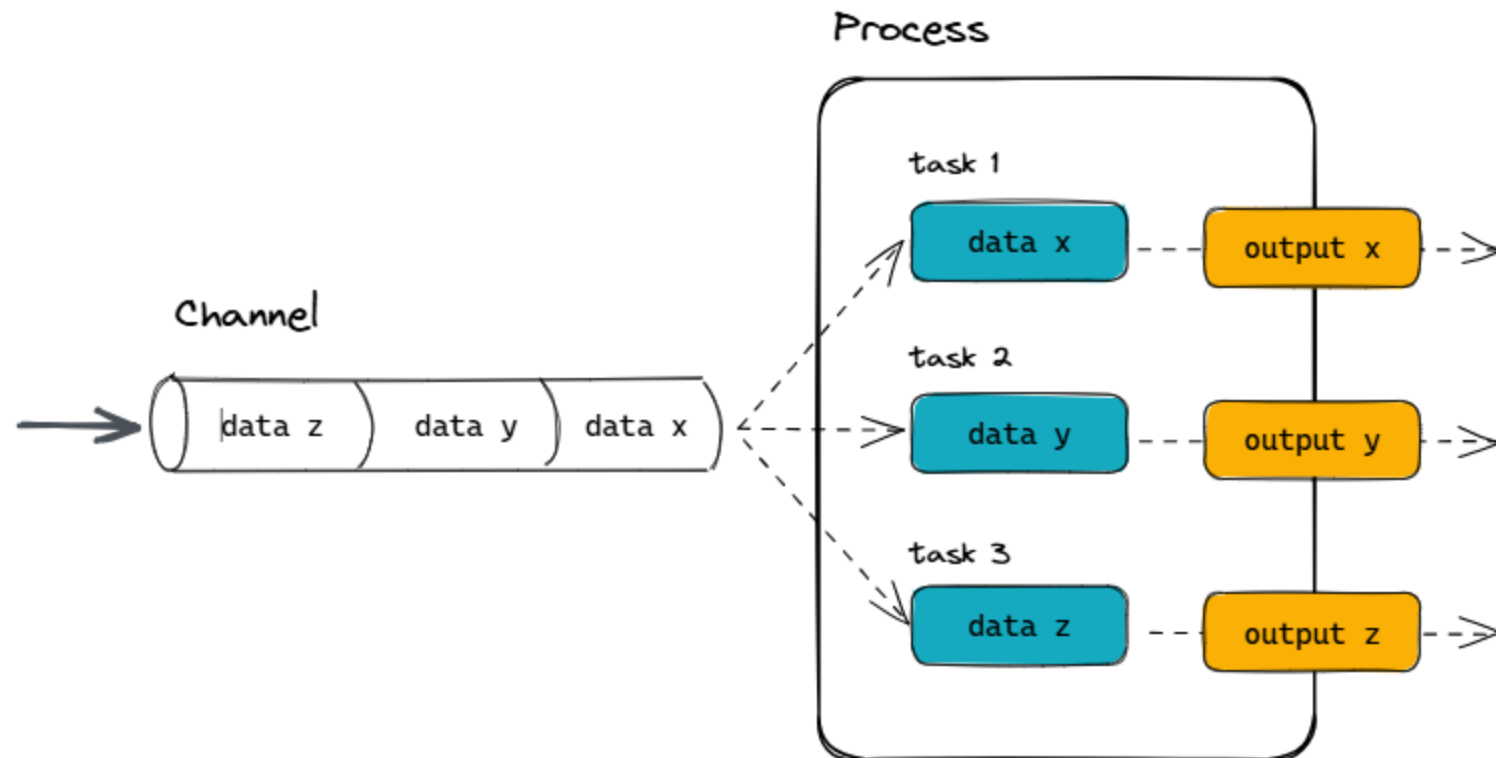
## What is nextflow?

- workflow orchestration engine, domain-specific language (in contrast to general-purpose language, e.g. python)
- easy to write data-intensive computational workflows
- extension of groovy which is a superset of Java
- core features:
  - portability and reproducibility
  - scalability of parallelization and deployment
  - integration of existing tools, systems, and industry standards

# How does it work?

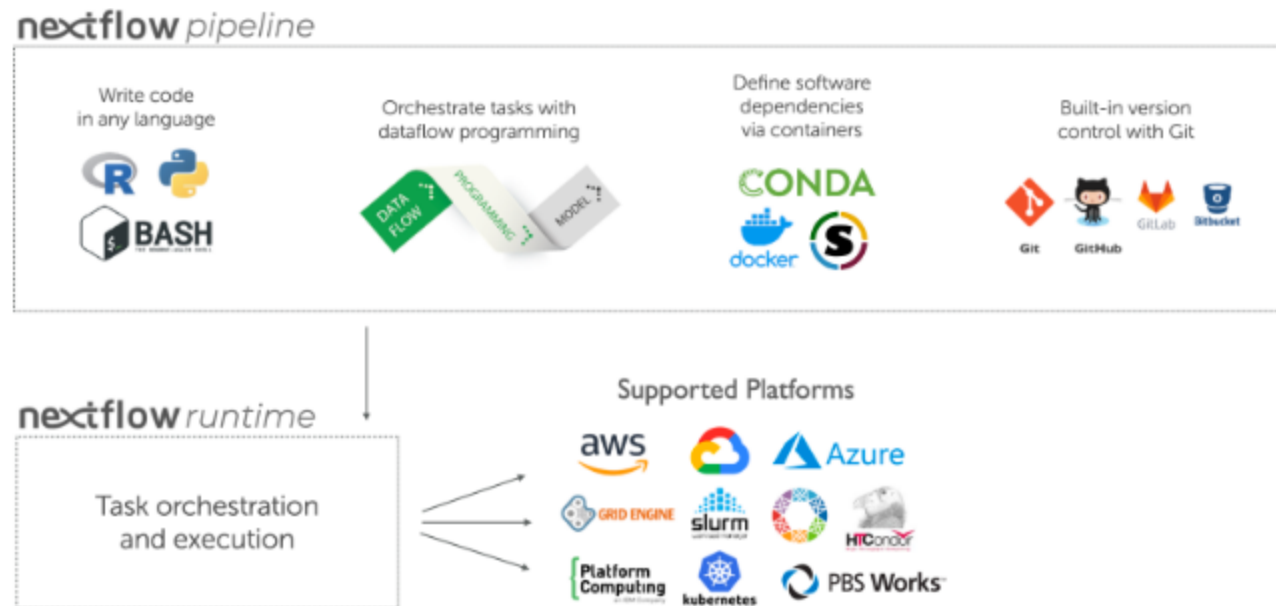
## Processes and channels

- different processes joined together - each written in any language that can be executed by Linux platform
- independently and isolated processes - not sharing common (writable) state
- communication via asynchronous first-in, first-out (FIFO) queues, called `channels`



# Execution abstraction

- process = *what* is executed  $\leftrightarrow$  executor = *how* it is executed
- provides abstraction between workflow's functional logic and underlying execution system/runtime
- workflow runs seamlessly on local computer, HPC cluster or cloud



# Let's get started

- go to [https://github.com/InPreD/25-06\\_bioinfo\\_ws\\_nextflow](https://github.com/InPreD/25-06_bioinfo_ws_nextflow)

The screenshot shows the GitHub interface for the repository `25-06_bioinfo_ws_unit_testing`, which is public. At the top, there are buttons for `Edit Pins`, `Watch` (1), `Fork` (0), and `Star` (0). Below the repository name, it shows `main` branch, `1 Branch`, and `0 Tags`. A search bar labeled `Go to file` and buttons for `Add file` and `<> Code` are visible. The file list on the left includes `.devcontainer` (fix: rm proxy), `LICENSE` (Initial commit), and `README.md` (Initial commit). The `README` file is selected, showing the title `25-06_bioinfo_ws_unit_testing` and the description `codespace template for unit testing workshop`. A dropdown menu is open over the `Code` button, showing options for `Local` and `Codespaces`. The `Codespaces` section lists `shiny space eureka` on the `main` branch with `No changes` and a `3d` ago timestamp. A button `Create a codespace on main` is highlighted. The right sidebar contains the `About` section with details like `codespace template for unit testing workshop`, `Readme`, `AGPL-3.0 license`, `Activity`, `Custom properties`, `0 stars`, `1 watching`, and `0 forks`. Below this are sections for `Releases` (No releases published, [Create a new release](#)), `Packages` (No packages published, [Publish your first package](#)), `Languages` (a bar chart showing `Dockerfile` at 100.0%), and `Suggested workflows` (Based on your tech stack).



- create a branch for your work:

```
$ git checkout -b unit-tests-<your name>
```

- create a new file `hello_world.nf`
- write a workflow which outputs a file containing "Hello World!"

```
#!/usr/bin/env nextflow

/*
 * Use echo to print 'Hello World!' to a file
 */
process sayHello {

    output:
        path 'output.txt'

    script:
        """
        echo 'Hello World!' > output.txt
        """
}

workflow {

    // emit a greeting
    sayHello()
}
```