

Bioinformatics session

3rd Annual workshop on
bioinformatics and variant
interpretation in InPreD

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

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 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 describes it as a "codespace template for unit testing workshop" and lists details: **Readme**, **AGPL-3.0 license**, **Activity**, **Custom properties**, **0 stars**, **1 watching**, and **0 forks**. It also includes links for **Report repository**, **Releases** (No releases published, [Create a new release](#)), **Packages** (No packages published, [Publish your first package](#)), and **Languages** (a bar chart showing **Dockerfile** at 100.0%). The **Suggested workflows** section is partially visible at the bottom.

First exercise

← →

25-06_bioinfo_ws_unit_testing [Codespaces: shiny space eureka]

👤

☰

EXPLORER

...

📁

🔍

🔗

🔧

📁

🐙

👤

⚙️

25-06_BIOINFO_WS_UNIT_TESTING [CODESPA...

> .devcontainer

🔑 LICENSE

📘 README.md

[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#
```

> GLIEDERUNG

> ZEITACHSE

<> 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
```

2. Nextflow