

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 repository page for '25-06_bioinfo_ws_unit_testing' by user 'marrip'. The repository is public and has 1 branch and 0 tags. A 'Codespaces' overlay is visible, showing a list of workspaces: 'Local' (fix: rm proxy), 'Codespaces' (Your workspaces in the cloud), and 'shiny space eureka' (On current branch, 3d ago, No changes). A button 'Create a codespace on main' is highlighted. The repository description is 'codespace template for unit testing workshop'. The right sidebar shows the 'About' section with links to Readme, AGPL-3.0 license, Activity, Custom properties, 0 stars, 1 watching, and 0 forks. The 'Releases' section shows 'No releases published' and a link to 'Create a new release'. The 'Packages' section shows 'No packages published' and a link to 'Publish your first package'. The 'Languages' section shows a bar chart with 'Dockerfile' at 100.0%.

25-06_bioinfo_ws_unit_testing Public

Edit Pins Watch 1 Fork 0 Star 0

main 1 Branch 0 Tags

Go to file Add file <> Code

marrip fix: rm proxy

.devcontainer fix: rm proxy

LICENSE Initial commit

README.md Initial commit

README AGPL-3.0 license

25-06_bioinfo_ws_unit_testing

codespace template for unit testing workshop

Local Codespaces

Codespaces Your workspaces in the cloud

On current branch

shiny space eureka 3d ...

main No changes

Create a codespace on main

Codespace usage for this repository is paid for by marrip.

About

codespace template for unit testing workshop

Readme

AGPL-3.0 license

Activity

Custom properties

0 stars

1 watching

0 forks

Report repository

Releases

No releases published

Create a new release

Packages

No packages published

Publish your first package

Languages

Dockerfile 100.0%

Suggested workflows

Based on your 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](#)):

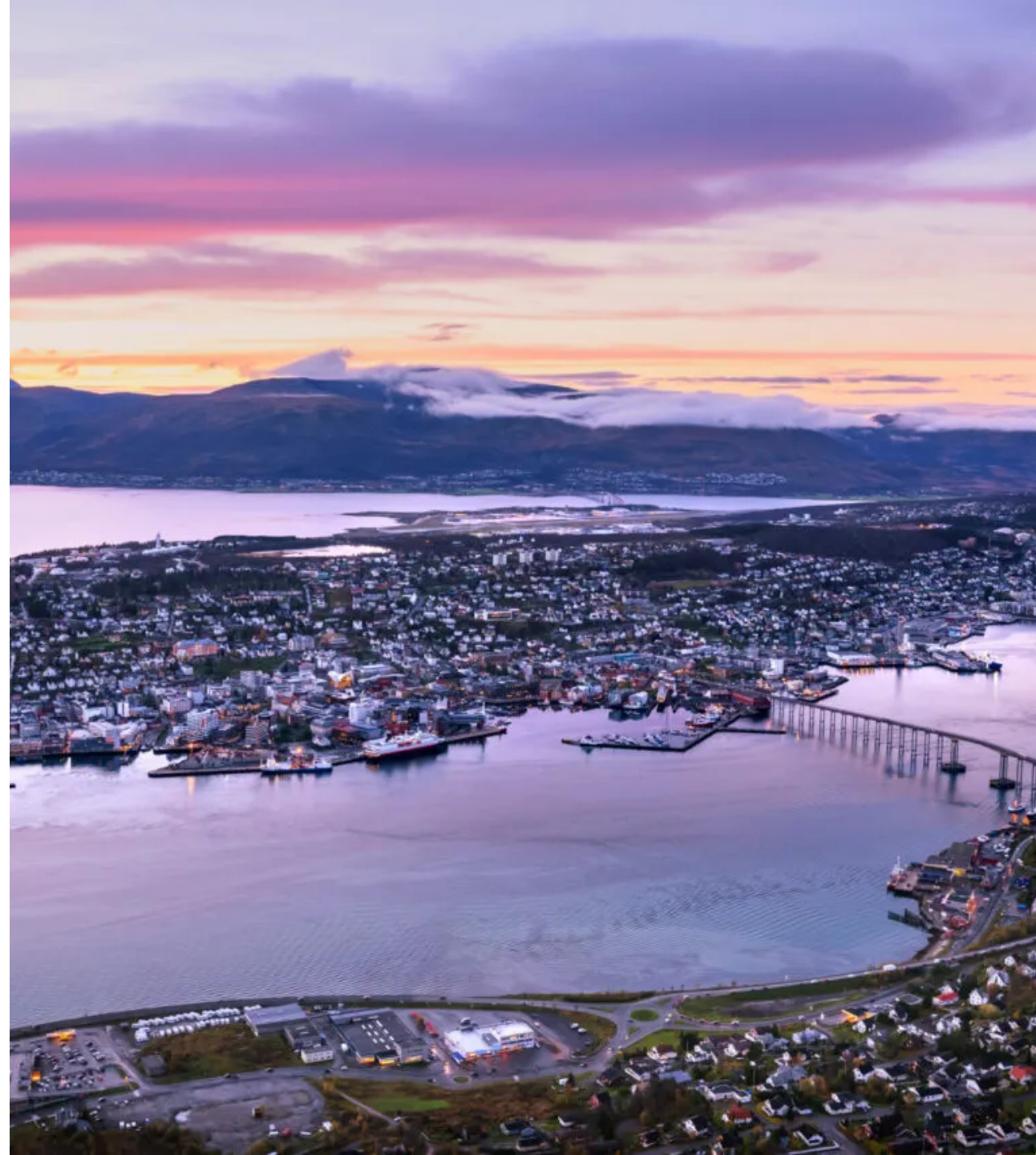
```
$ git add third/tests/third_test.py  
$ git commit -m "test: <your commit message>"
```

- and we push them to GitHub:

```
$ git push
```

Thank you for your attention!

Day 1 done!



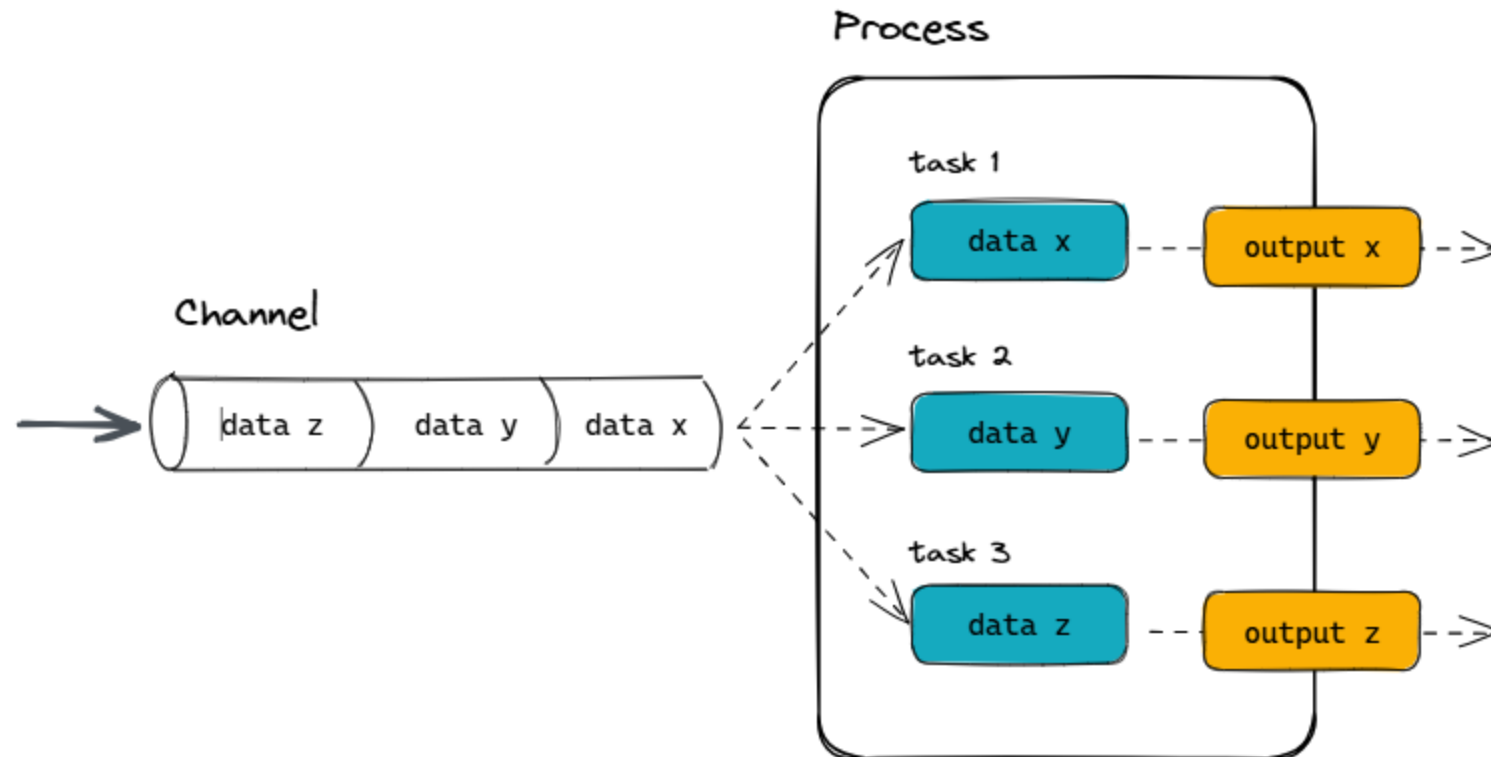
2. Nextflow

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

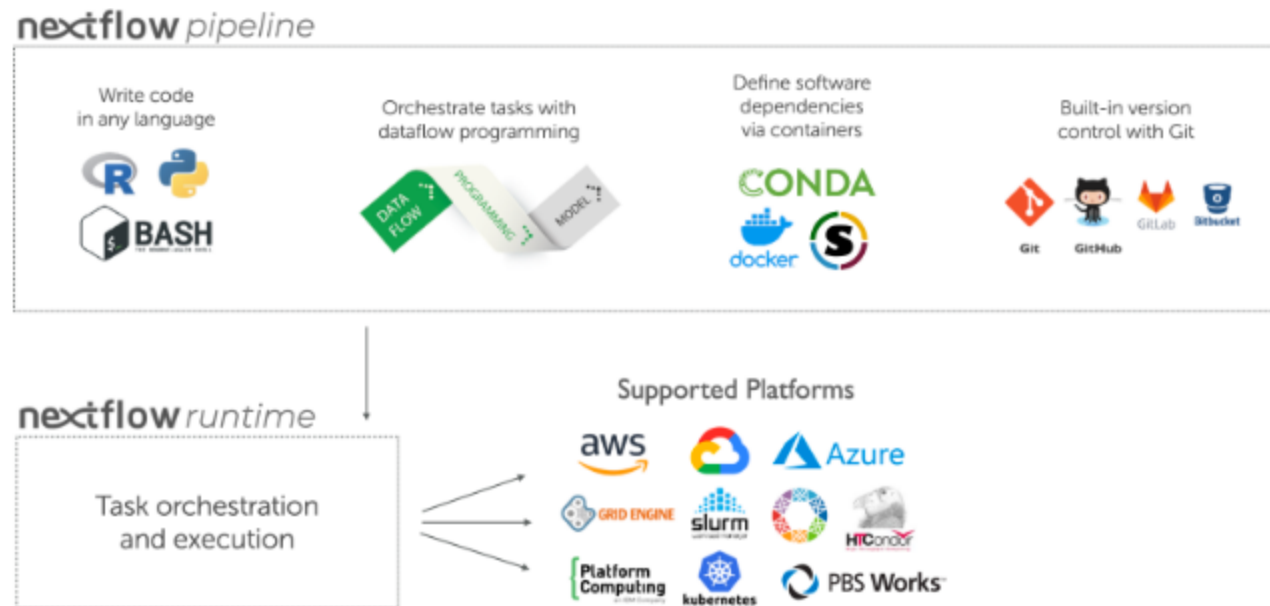
Processes and channels

- 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 <-> 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

The screenshot shows the GitHub interface for the repository **25-06_bioinfo_ws_nextflow**, 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 'Go to file' and buttons 'Add file' and '<> Code' are visible. A file list on the left includes '.devcontainer', '.gitignore', 'LICENSE', and 'README.md'. A 'Codespaces' overlay is open, showing 'Local' and 'Codespaces' tabs. The 'Codespaces' tab displays 'Your workspaces in the cloud' with a '+ ...' button and a 'Create a codespace on main' button. It also shows a workspace 'miniature fishstick' on the 'main' branch with 'Uncommitted changes' and a duration of '1h'. A note at the bottom of the overlay states 'Codespace usage for this repository is paid for by marrip.'. The main content area shows the 'README' file with the title '25-06_bioinfo_ws_nextflow' and the description 'codespace template for nextflow workshop'. The right sidebar contains sections for 'About' (codespace template for nextflow workshop), 'Releases' (No releases published, Create a new release), 'Packages' (No packages published, Publish your first package), and 'Languages' (Dockerfile 100.0%).

- create a branch for your work

```
$ git checkout -b nextflow-<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()
}
```

- try to run it

```
$ nextflow run hello_world.nf
```

```
root@52abc3ecdd95 /w/25-06_bioinfo_ws_nextflow (main)# nextflow run hello_world.nf
Nextflow 25.04.1 is available - Please consider updating your version to it

NEXTFLOW ~ version 24.10.4

Launching `hello_world.nf` [boring_rubens] DSL2 - revision: 595ea09581

executor > local (1)
[33/7106de] process > sayHello [100%] 1 of 1 ✓
```

- check if you can find `work/33/7106de/output.txt`
- explore the other files that are in `work/33/7106de/`

- add a directory to which results should be published

```
process sayHello {  
    publishDir 'results', mode: 'copy'  
    output:  
        path 'output.txt'  
    ...  
}
```

- add a greeting variable

```
process sayHello {  
    publishDir 'results', mode: 'copy'  
  
    input:  
        val greeting  
  
    output:  
        path 'output.txt'  
  
    script:  
        """  
        echo '$greeting' > output.txt  
        """  
}  
  
workflow {  
    // emit a greeting  
    sayHello(params.greeting)  
}
```

```
$ nextflow run hello_world.nf --greeting 'Heisann!'
```

- move the `params.greeting` into the `hello_world.nf` :

```
params.greeting = 'Heisann!'

process sayHello {
    publishDir 'results', mode: 'copy'
    ...
}
```

```
$ nextflow run hello_world.nf
```

```
$ nextflow run hello_world.nf --greeting 'Hejsan!'
```

- transform greetings into a channel

```
process sayHello {  
    input:  
        val greeting  
  
    output:  
        stdout  
  
    script:  
    """  
    echo '$greeting'  
    """  
}  
  
workflow {  
    greeting_ch = Channel.of('Alo', 'Salut', 'Sunt eu').view()  
    // emit a greeting  
    sayHello(greeting_ch) | view  
}
```

```
$ nextflow run hello_world.nf -ansi-log false # run it several times
```

Channels

- processes communicate through channels
- two major properties:
 - Sending a message is an asynchronous (i.e. non-blocking) operation
 - Receiving a message is a synchronous (i.e. blocking) operation
- please refer to the [nextflow docs about channels](#)

- create a `.csv` file containing our greetings which will serve as input

```
Alo  
Salut  
Sunt eu
```

- create channel from the file using the channel factory `fromPath()`

```
workflow {  
  
    greeting_ch = Channel.fromPath(params.greeting).view()  
    // emit a greeting  
    sayHello(greeting_ch) | view  
}
```

```
$ nextflow run hello_world.nf -ansi-log false --greeting greetings.csv
```

- try to manipulate a channel by using the operators `splitCsv()` and `map()`

```
workflow {  
    greeting_ch = Channel.fromPath(params.greeting)  
        .view( it -> "Before splitCsv: $it" )  
        .splitCsv()  
        .view( it -> "After splitCsv: $it" )  
        .map( item -> item[0] )  
        .view( it -> "After map: $it" )  
    // emit a greeting  
    sayHello(greeting_ch) | view  
}
```

```
$ nextflow run hello_world.nf -ansi-log false --greeting greetings.csv
```

- please refer to the [nextflow docs about channel operators](#)

- add a second process

```
process convertToUpper {  
    input:  
        val lower  
  
    output:  
        stdout  
  
    script:  
        ""  
        echo '$lower' | tr '[a-z]' '[A-Z]'  
        ""  
}
```

- include the process in the workflow and link it to the first one

```
workflow {  
  
    greeting_ch = Channel.fromPath(params.greeting)  
        .view( it -> "Before splitCsv: $it" )  
        .splitCsv()  
        .view( it -> "After splitCsv: $it" )  
        .map( item -> item[0] )  
        .view( it -> "After map: $it" )  
  
    // emit a greeting  
    sayHello(greeting_ch) | view  
    convertToUpper(sayHello.out) | view  
}
```

```
$ nextflow run hello_world.nf -ansi-log false --greeting greetings.csv
```

- run first process inside a container

```
process sayHello {  
    container 'ubuntu:24.04'  
    input:  
        val greeting  
  
    output:  
        stdout  
  
    script:  
        """  
        echo '$greeting'  
        """  
}
```

- add `nextflow.config` containing

```
docker.enabled = true
```

- please refer to the [nextflow docs on configuration](#)

- check your docker images

```
$ docker images
```

- now run

```
$ nextflow run hello_world.nf -ansi-log false --greeting greetings.csv
```

- check your docker images again

```
$ docker images
```

- restructure your workflow a bit by moving the processes in a `modules/` subfolder
- each process should get its own file, e.g. `modules/sayHello.nf`

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

process sayHello {

    input:
        val greeting

    output:
        stdout

    script:
        """
        echo '$greeting'
        """

}
```

- include the modules in the header of your workflow file

```
include { sayHello      } from './modules/sayHello.nf'
include { convertToUpper } from './modules/convertToUpper.nf'

workflow {
    ...
}
```

```
$ nextflow run hello_world.nf -ansi-log false --greeting greetings.csv
```


nf-core

- diverse project spread across many groups (Seqera, SciLifeLab Sweden, Centre for Genomic Regulation etc.)
- community effort to collect a curated set of analysis pipelines built using nextflow
- standardised set of best practices, guidelines, and templates
- modular, scalable, and portable pipelines - researchers can easily adapt and execute them using own data and compute resources
- open development, testing, and peer review -> pipelines are robust, well-documented, and validated against real-world datasets

- 113 nf-core pipelines (October 2024):
 - 68 released
 - 32 under development
 - 13 archived
- please refer to the [nf-core website](#) for more information and resources

- create a pipeline template using `nf-core` tooling

```
$ nf-core pipelines create
```

- in the tui we choose `Let's go` > `Custom` (**OBS! Name should not contain -**)

Basic details

GitHub organisation

Workflow name

inpred

test_pipeline

A short description of your pipeline.

exciting pipeline for testing

Name of the main author / authors

@marrip

Back Next

- unselect `Toggle all features` and select the following:
 - `Add configuration files`
 - `Use code linters`
 - `Use fastqc`
 - `Use nf-core components`
 - `Use nf-schema`
 - `Add testing profiles`
- continue with `Continue` > `Finish` > `Continue` > `Finish without creating a repo` > `Close`
- take a look at the output

- **stubbing** = quickly prototype the workflow logic without using the real commands; comparable to dry-run
- prepare for a stub run by adding mock fastq file to `assets/`

```
$ touch assets/sample1_R1.fastq.gz assets/sample1_R2.fastq.gz assets/sample2_R1.fastq.gz
```

- update `assets/samplesheet.csv` like so

```
sample,fastq_1,fastq_2  
SAMPLE_PAIRED_END,assets/sample1.fastq_R1.gz,assets/sample1_R2.fastq.gz  
SAMPLE_SINGLE_END,assets/sample2.fastq_R1.gz
```

- run your first nf-core pipeline

```
$ nextflow run . -stub --input assets/samplesheet.csv --outdir results # -ansi-log false
```

- add the nf-core module `bwa/mem`

```
$ nf-core modules install bwa/mem
```

- add it to `workflows/test_pipeline.nf`

```
include { BWA_MEM } from '../modules/nf-core/bwa/mem/main'
include { FASTQC } from '../modules/nf-core/fastqc/main'
...
    ch_versions = ch_versions.mix(FASTQC.out.versions.first())

    //
    // MODULE: Run bwa mem
    //
    BWA_MEM (
        ch_samplesheet,
        [[], []],
        [[], []],
        []
    )
```

- in the stub section of `modules/nf-core/bwa/mem/main.nf`, change the version command string to

```
"""  
...  
cat <<-END_VERSIONS > versions.yml  
"${task.process}":  
  bwa: mock  
  samtools: mock  
END_VERSION  
"""
```

```
$ nextflow run . -stub --input assets/samplesheet.csv --outdir results # -ansi-log false
```


- try to add other modules from `nf-core/modules`, e.g. `trimmomatic` or `cutadapt`, and link them them to `bwa/mem`

Thank you for your attention!

Day 2 done!

