Unterstützt von / Supported by

Alexander von Humboldt
Stiftung / Foundation

Good practices tutorial: CI/CD & Testing

Manuel Guth

FTAG Algo tutorial - Good code practices

14.04.2022

ATLAS
EXPERIMENT

UNIVERSITÉ
DE GENÈVE

FACULTÉ DES SCIENCES

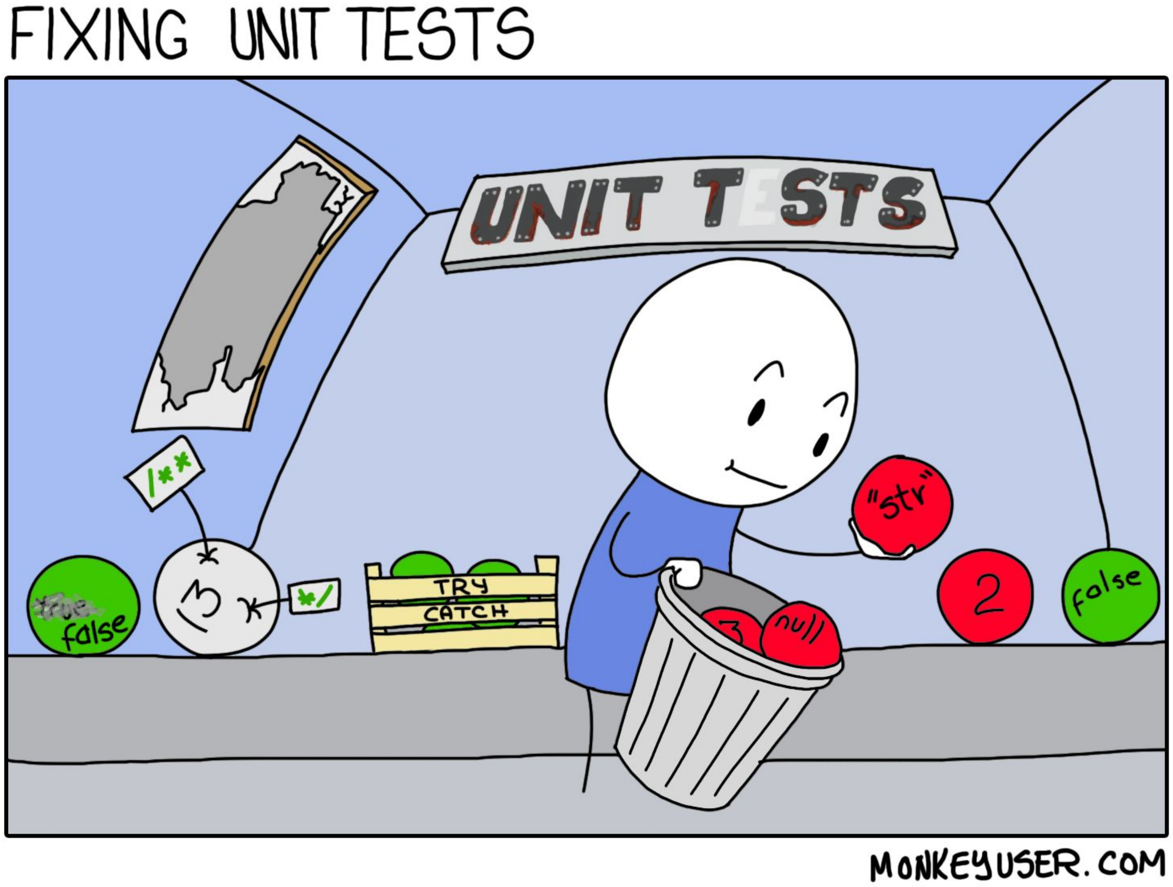
Overview

- Testing
  - unit tests
  - integration tests
  - simply running job in CI
- Auxiliary material
  - Setup of CI tests
    - What is CI?
    - Configuring a gitlab CI → default setups
    - Job configurations
    - Building docker images on (CERN) gitlab

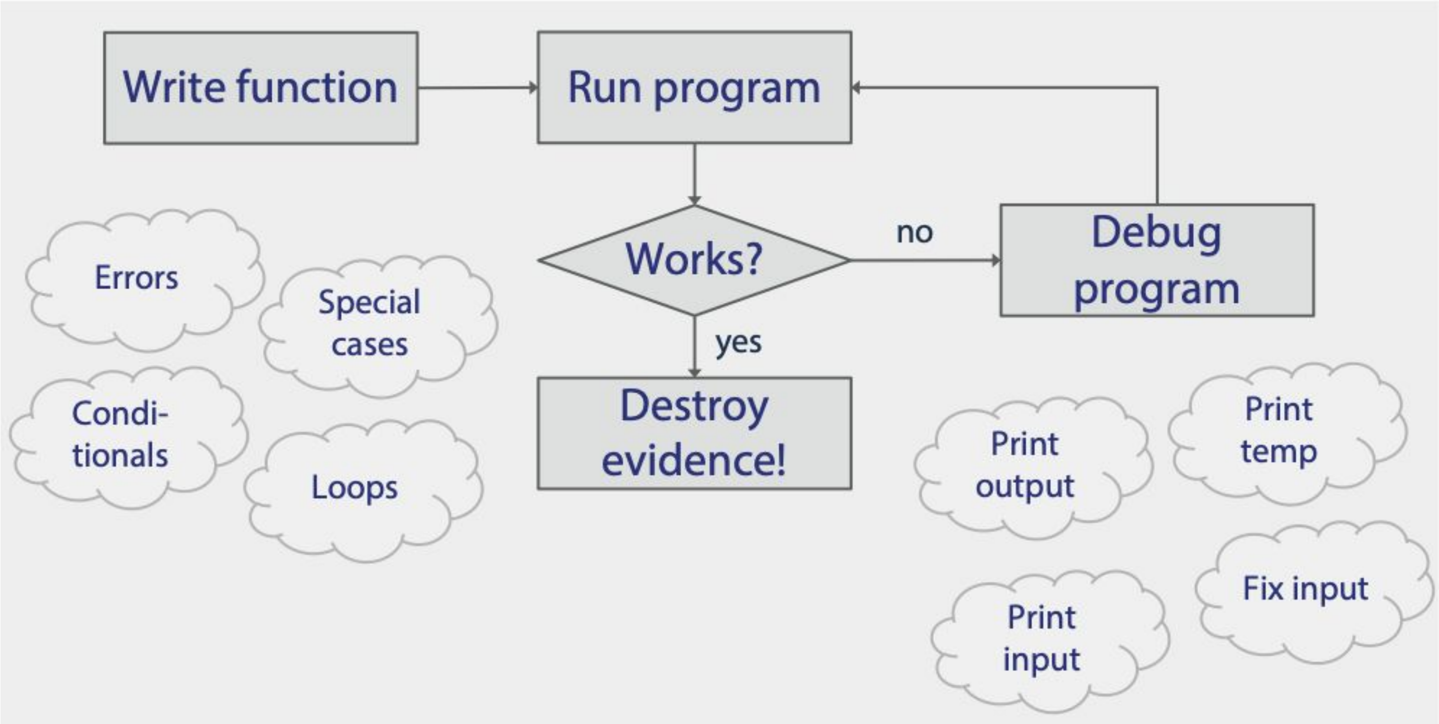
Testing

With material from Michael Koenig

Unit Tests


FIXING UNIT TESTS
UNIT TESTS
/**
*/
~~true~~
false
TRY
CATCH
"str"
2
false
3
null
MONKEYUSER.COM

Typical workflow in Science (physics)

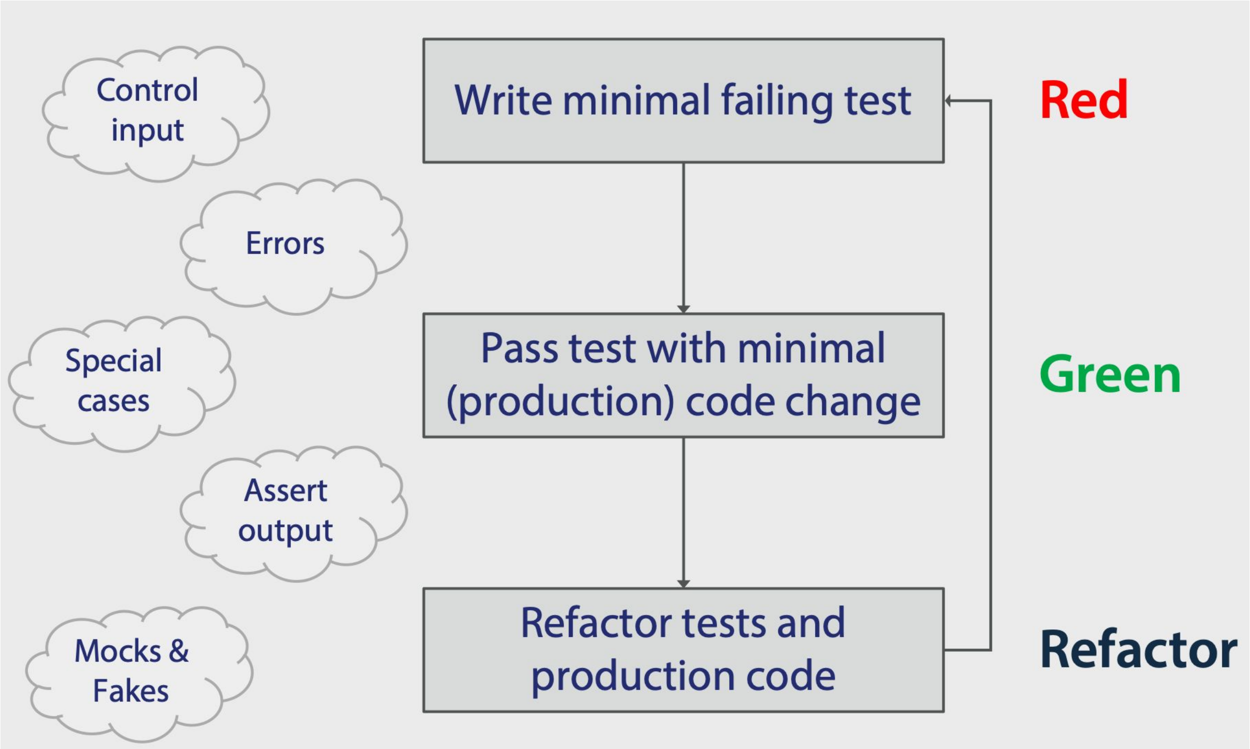


Michael Koenig

Why we need tests?

- Code which is not being tested has several issues
  - Unclear behavior
    - Little changes could introduce bugs which are undetected
  - Manual testing not optimal
    - Often not done in systematic way
    - Mostly not reproducible (e.g. only manually changed values w/o documenting them)
    - Lots of effort redoing it
  - Not my code
    - Difficult to quickly assess code of other people to judge if it does what it should
  - “That was me?”
    - After some time one does not remember everything anymore even about your own code
- Unit tests are written when developing the code
  - Code more robust and often more performant (addressing the problem in different test scenarios)
  - Automation allows others to change your code without losing targeted functionality

Typical workflow for test-driven development



Control input
Errors
Special cases
Assert output
Mocks & Fakes
Write minimal failing test
Red
Pass test with minimal (production) code change
Green
Refactor tests and production code
Refactor

Michael Koenig

Testing boosts your code

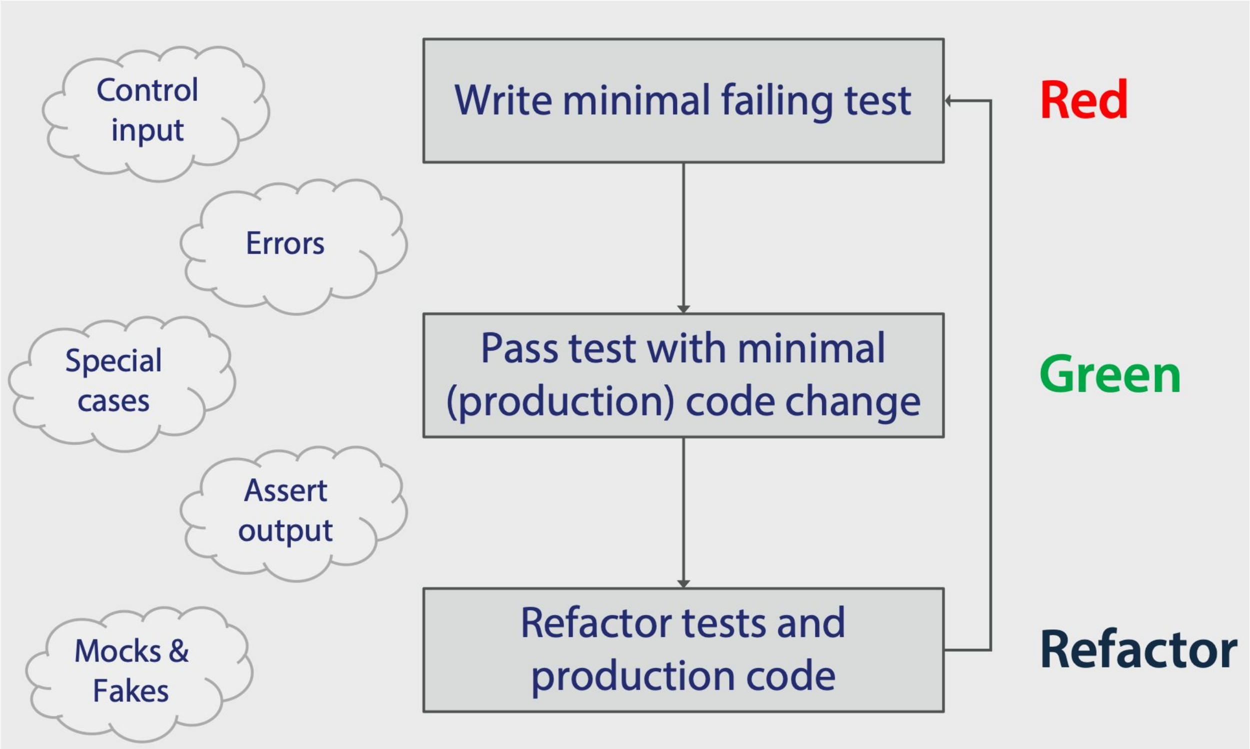
- Tiny steps?
  - Not necessarily
  - Write failing test
  - Write obvious implementation
- TDD lets you work as fast as you can
- Impact on code
  - Modular design
  - Cleaner code
  - Less bugs
- Impact on tests
  - Full automation
  - 100% coverage
  - Executable specs

YES!
TDD = Test Driven Development
Widely used in the community

TDD boosts your work life

- Steady sense of progress
- Ease of mind
- Courage

Typical workflow for test-driven development



Control input
Errors
Special cases
Assert output
Mocks & Fakes
Write minimal failing test
Pass test with minimal (production) code change
Refactor tests and production code
Red
Green
Refactor

Michael Koenig

Red: Write minimal failing test

- Minimal
  - Prevents complexity
- Execute all tests
  - Prevents slow tests
- Assert new test fails
  - Prevent inactive tests
  - Prevents bugs in tests
  - Prevent complexity

Minimal means:

- Missing import
- Missing class
- Missing function
- One assertion a time
- Simple to complex
  - Error cases first
  - Corner cases next
  - General behaviour last

Green: Pass test with minimal change

- Minimal
  - Prevents complexity
- Execute all tests
  - Prevents slow tests
- Assert all test succeed
  - Prevents bugs in code
  - Prevent bugs in tests

Minimal means:

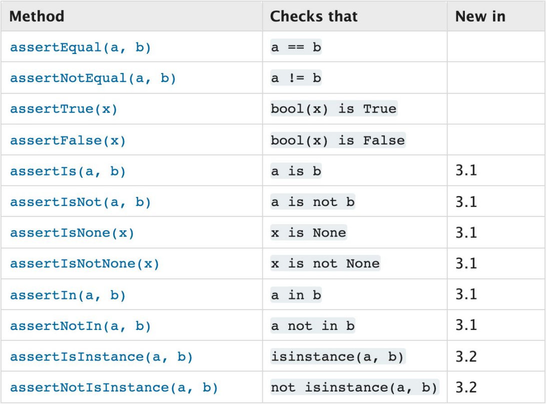
- Add file stub
- Add class stub
- Add function stub
- Unconditionally raise
- Hard-coded results
- Correctly sized results
- Defer conditionals
- Defer loops

Refactor: Clean up test/production code

- Remove superseded tests
    - Better signal/noise ratio
- Clean code principles
    - Reduce complexity
- Execute all tests
    - Prevents slow tests
    - Prevents refactoring bugs
    - Prevents brittle tests

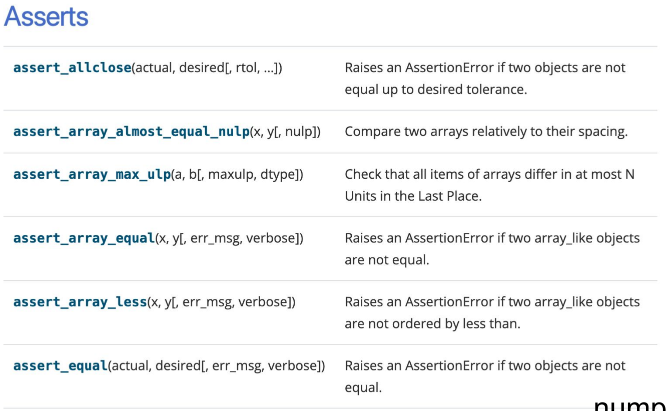
Several libraries can help writing unit tests

- Standard libraries
  - Pytest
  - Unittest
- Several libraries have their own assertation implementation for unit tests
  - e.g. numpy: https://numpy.org/doc/stable/reference/routines.testing.html



Method	Checks that	New in
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2

unittest



Asserts

assert_allclose(actual, desired[, rtol, ...])	Raises an AssertionError if two objects are not equal up to desired tolerance.
assert_array_almost_equal_nulp(x, y[, nulp])	Compare two arrays relatively to their spacing.
assert_array_max_ulp(a, b[, maxulp, dtype])	Check that all items of arrays differ in at most N Units in the Last Place.
assert_array_equal(x, y[, err_msg, verbose])	Raises an AssertionError if two array_like objects are not equal.
assert_array_less(x, y[, err_msg, verbose])	Raises an AssertionError if two array_like objects are not ordered by less than.
assert_equal(actual, desired[, err_msg, verbose])	Raises an AssertionError if two objects are not equal.

numpy

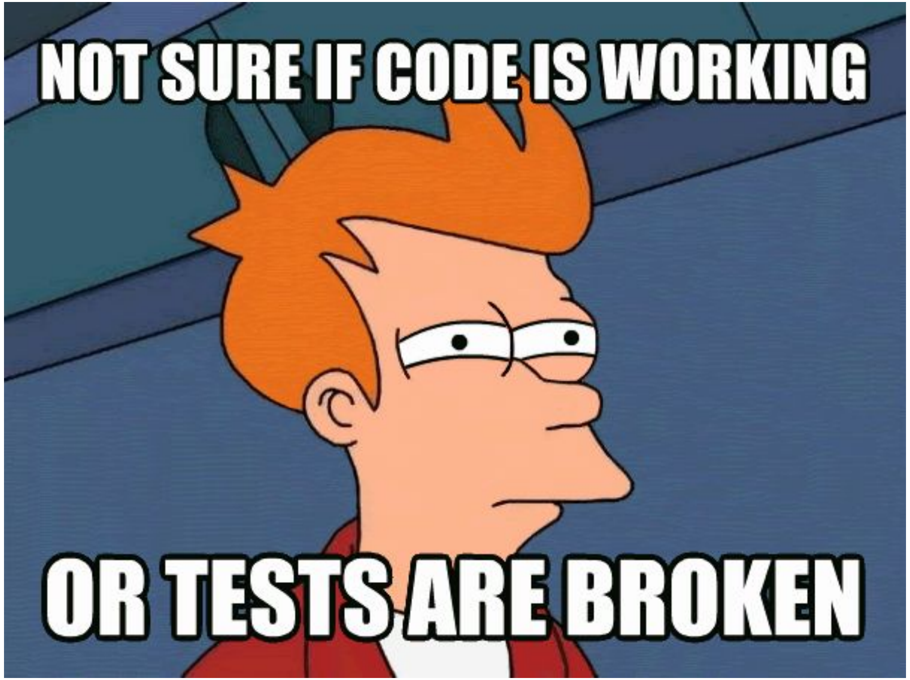
Integration tests

"Hm, worked in tests when I poured water directly into drain"

What is the difference w.r.t unit tests?

- Unit tests check single functions
- Integration tests use “real-life” setup
  - e.g. running a NN traning
- Implementation
  - Possible to do also with unittest/pytest setup
    - Rather complicated
  - Simply running in the gitlab pipeline (CI)
    - We will focus on this option

Hands-on



Hands-on Setup

Use the tutorial repository

- CERN gitlab
    - `https://gitlab.cern.ch/mguth/good-code-practices`

Hands on unit tests - Part I

A (failing) example is given for palindromes [inspired by this tutorial].

To run the unit tests, you need to go into the python-testing folder

```
cd python-testing
```

and then you can run the tests via

```
pytest -v test_palindrome.py
```

As a first exercise, please fix the unit tests by adapting the palindrome code.
Are all the tests making sense?

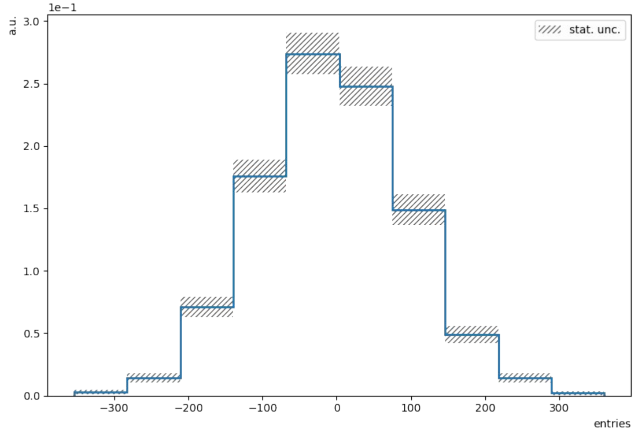
Hands on tests - Part II

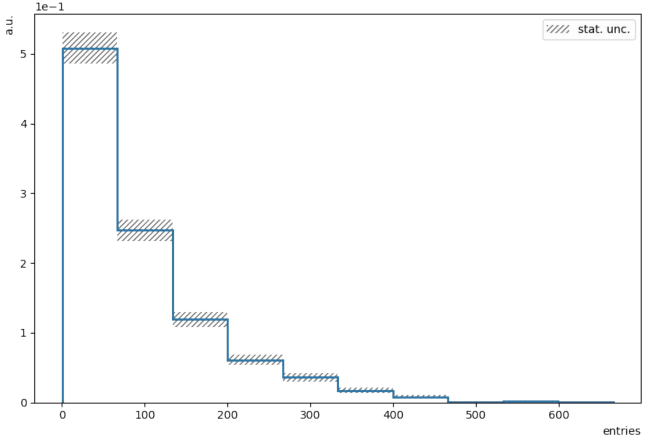
- three functions (classes)

  `Generate_data, histogram` and `plot_histogram`

  already predefined in folder `mymodule`.
- Write functionalities of these 3 functions
- Write unit tests for `generate_data` and `histogram`.
  - Write your unit tests in the `test_unit_mymodule.py` file.

- Write an integration test, to test the full chain
  - This will be implemented in `integration_test_my_module.py`
  - Add to the gitlab CI in the `.gitlab-ci.yml` file.





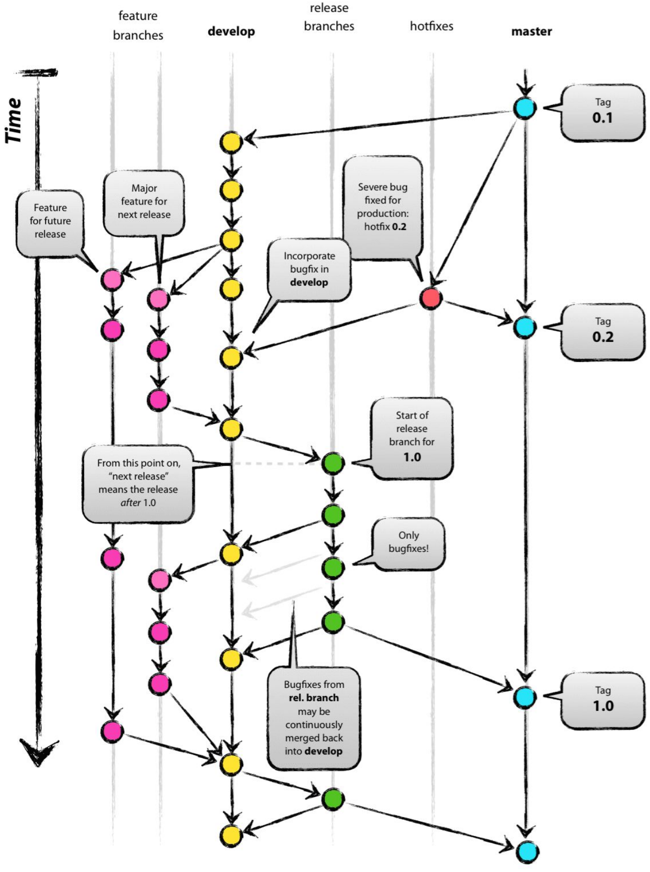
Auxiliary Material

Continuous integration & deployment (Material from K. Zoch & M. Guth - RODEM tutorial)

Recap of the branching model

General assumption: we all use git for all our projects!

- Successful collaboration → briefly touched branching models in git presentation last week.
- It doesn’t have to be as complex as this ‘git flow’ model!
- Still, collaborating with multiple people means:
  - You are not the only person that needs to understand your code! → Documentation is extremely important.
  - Work with & review code you haven’t written yourself
    - High code quality & modern standards
    - Uniform & consistent style (linters!)
  - Potential for merge conflicts
  - Thorough, automated tests of every part of the code will help to spot problems early on.


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Incorporate bugfix in develop
Severe bug fixed for production: hotfix 0.2
Tag 0.2
Start of release branch for 1.0
From this point on, “next release” means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

Continuous integration

A great way out of it: continuous integration – master/main branch always contains a "working version" of the code.

How to achieve that:

- Don't ever allow `git push upstream master`!
- All changes only ever come through merge requests.
- If developer pool is large: only let code maintainers push to the upstream repository. Merge requests from forks.
- Test suite to verify changes from a merge request:
  - "Would the application still show expected behaviour if these changes were merged?"
  - Require tests of any new bits of code.

→ More about tests in a few minutes by Manuel!

YOU SAID YOU THOUGHT YOU MERGED CHANGES INTO A FEATURE BRANCH
maury
THE GIT LOG SHOWS THAT WAS A LIE
makeameme.org

Configuring a gitlab CI

Continuous integration tests can be set up easily on gitlab. Few requirements:

- A good idea / plan **what** you want to test!
  - Individual functions, methods, classes, modules of your code.
  - Configuring & building your code on various architectures / systems (for compile-based languages).
  - Testing your code against various dependencies (e.g. support for earlier python versions).
  - A "real-life" test run of your code, e.g. a NN training on a mini dataset.
  - Validity of your files (e.g. yaml/json/python syntax, executability of your README code blocks).
  - ...
- A **configuration file** that defines your tests: `./.gitlab-ci.yml` .
- **Runners** available on your gitlab installation and for your repository!
  - Remember that these tests need to be run on resources – which are part of the gitlab installation.
  - On https://gitlab.cern.ch: very large suite of runners available, quite powerful.
  - On https://gitlab.com: generic runners available, CPU time per user very limited if no subscription.
  - On https://gitlab.unige.ch: unfortunately no runners installed …

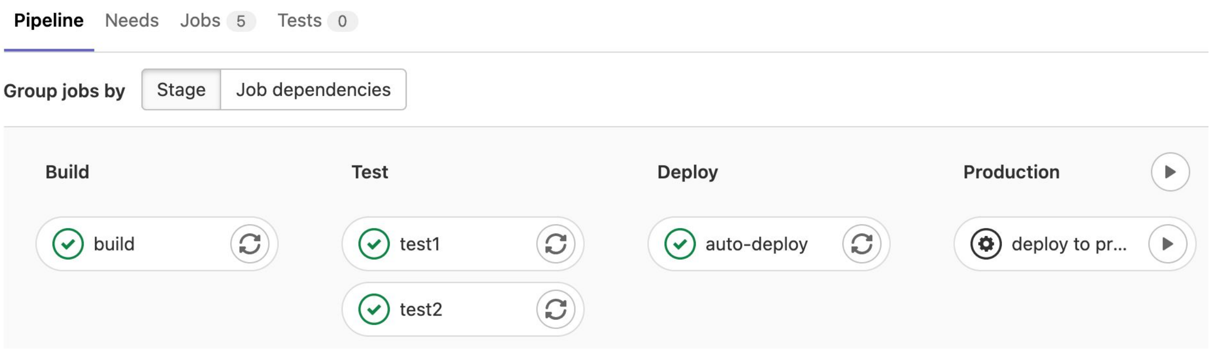
Configuring a gitlab CI

In the end: a CI runner = a resource to run automated tasks under certain conditions

CI runners can do so much more than just testing:

- Trigger some actions on a merge request (gitlab has a python API!)
  - Asking for review
  - Automated approval under certain conditions
  - Automated build or coverage reports
- Build and deploy docker images that ship your code
- Bundle and deploy your code somewhere (e.g. python eggs, c++ binaries)
- Deploy your code documentation to a static website
- …

Stages of a pipeline



Before starting to write individual jobs, decide which job stages you need, e.g.:

```
stages:
  - linting
  - unit_test
  - integration_test
  - deploy
```

Job configuration

```yaml
pylint:
  stage: linting
  image: python:3.7-slim
  script:
    - mkdir -p test_results/
    - pip install pylint
    - pylint my_script.py | tee test_results/linting.log
  artifacts:
    when: always
    paths:
      - test_results/
  rules:
    - if: $CI_COMMIT_BRANCH != ""
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
```

Job configuration

```
pylint:
  stage: linting
  image: python:3.7-slim
  script:
    - mkdir -p test_results/
    - pip install pylint
    - pylint my_script.py | tee test_results/linting.log
  artifacts:
    when: always
    paths:
      - test_results/
  rules:
    - if: $CI_COMMIT_BRANCH != ""
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
```

At which stage of the pipeline should this job run?

Job configuration

```
pylint:
  stage: linting
  image: python:3.7-slim
  script:
    - mkdir -p test_results/
    - pip install pylint
    - pylint my_script.py | tee test_results/linting.log
  artifacts:
    when: always
    paths:
      - test_results/
  rules:
    - if: $CI_COMMIT_BRANCH != ""
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
```

Use a docker image for this job

Job configuration

What actually gets executed?

```
pylint:
  stage: linting
  image: python:3.7-slim
  script:
    - mkdir -p test_results/
    - pip install pylint
    - pylint my_script.py | tee test_results/linting.log
  artifacts:
    when: always
    paths:
      - test_results/
  rules:
    - if: $CI_COMMIT_BRANCH != ""
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
```

Job configuration

```
pylint:
  stage: linting
  image: python:3.7-slim
  script:
    - mkdir -p test_results/
    - pip install pylint
    - pylint my_script.py | tee test_results/linting.log
  artifacts:
    when: always
    paths:
      - test_results/
  rules:
    - if: $CI_COMMIT_BRANCH != ""
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
```

Save “artifacts” (i.e. something that gets produced by the job) under certain conditions

Job configuration

```
pylint:
  stage: linting
  image: python:3.7-slim
  script:
    - mkdir -p test_results/
    - pip install pylint
    - pylint my_script.py | tee test_results/linting.log
  artifacts:
    when: always
    paths:
      - test_results/
  rules:
    - if: $CI_COMMIT_BRANCH != ""
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
```

Only run this job if one of the specified conditions is true

Things to specify for a job

- **Rules** – only run on: merge requests, master/main, tags, after manual trigger, …
- **Dependencies:** only run after another job has run
- A **default job**, e.g. to specify some common behaviour before/after a job has run
- **Variables:** similar to environment variables, something that needs to be picked up during a job
- **Image:** to decide a job should be run in a Docker image (not in OS environment of the runner)
- **External secrets:** secure variables, e.g. tokens, can also be stored within a repository to remain hidden (e.g. they cannot be printed in a CI job)
- … (and many more)

https://docs.gitlab.com/ee/ci/yaml/

Build a docker image in gitlab

There is a docker image to build docker images in the CI:

```
build:
  stage: build
  image:
    name: gcr.io/kaniko-project/executor:debug
    entrypoint: [""]
  script:
    - echo "{\"auths\":{\"${CI_REGISTRY}\":{\"auth\":\"$(printf "%s:%s"
"${CI_REGISTRY_USER}" "${CI_REGISTRY_PASSWORD}" | base64 | tr -d '\n')\"}}}" >
/kaniko/.docker/config.json
    - >-
      /kaniko/executor
      --context "${CI_PROJECT_DIR}"
      --dockerfile "${CI_PROJECT_DIR}/Dockerfile"
      --destination "${CI_REGISTRY_IMAGE}:${CI_COMMIT_TAG}"
  rules:
    - if: $CI_COMMIT_TAG
```

Build a docker image in gitlab

There is a docker image to build docker images in the CI:

```
build:
  stage: build
  image:
    name: gcr.io/kaniko-project/executor:debug
    entrypoint: [""]
  script:
    - echo "{\"auths\":{\"${CI_REGISTRY}\":{
"${CI_REGISTRY_USER}" "${CI_REGISTRY_PASSWOR
/kaniko/.docker/config.json
    - >-
      /kaniko/executor
      --context "${CI_PROJECT_DIR}"
      --dockerfile "${CI_PROJECT_DIR}/Dockerfile"
      --destination "${CI_REGISTRY_IMAGE}:${CI_COMMIT_TAG}"
  rules:
    - if: $CI_COMMIT_TAG
```

There is also a CERN image:
gitlab-registry.cern.ch/ci-tools/docker-image-builder

Build a docker image in gitlab

There is a docker image to build docker images in the CI:

```
image_build:
  stage: build
  image:
    name: gcr.io/kaniko-project/executor:debug
    entrypoint: [""]
  script:
    - echo "{\"auths\":{\"${CI_REGISTRY}\":{
"${CI_REGISTRY_USER}" "${CI_REGISTRY_PASSWOR
/kaniko/.docker/config.json
    - >-
      /kaniko/executor
      --context "${CI_PROJECT_DIR}"
      --dockerfile "${CI_PROJECT_DIR}/Dockerfile"
      --destination "${CI_REGISTRY_IMAGE}:${CI_COMMIT_TAG}"
  rules:
    - if: $CI_COMMIT_TAG
```

Runs only when a new git commit is created. Then creates ${IMAGE}:${COMMIT_TAG}

Build a docker image in gitlab

Can easily extend this to more rules:

Have no destination by default (i.e. just building)

```
image_build:
    [...]
    variables:
      DESTINATION_FLAG: "--no-push"
    rules:
      - if: $CI_PIPELINE_SOURCE == "merge_request_event"
      - if: $CI_COMMIT_REF_NAME == "master"
        variables:
          DESTINATION_FLAG: "--destination ${CI_REGISTRY_IMAGE}:latest"
      - if: $CI_COMMIT_TAG
        variables:
          DESTINATION_FLAG: "--destination ${CI_REGISTRY_IMAGE}:${CI_COMMIT_TAG}"
```

Build a docker image in gitlab

Can easily extend this to more rules:

Start destination-less build for all merge requests

```
image_build:
    [...]
    variables:
      DESTINATION_FLAG: "--no-push"
    rules:
      - if: $CI_PIPELINE_SOURCE == "merge_request_event"
      - if: $CI_COMMIT_REF_NAME == "master"
        variables:
          DESTINATION_FLAG: "--destination ${CI_REGISTRY_IMAGE}:latest"
      - if: $CI_COMMIT_TAG
        variables:
          DESTINATION_FLAG: "--destination ${CI_REGISTRY_IMAGE}:${CI_COMMIT_TAG}"
```

Build a docker image in gitlab

Can easily extend this to more rules:

```
image_build:
  [...]
  variables:
   DESTINATION_FLAG: "--no-push"
  rules:
   - if: $CI_PIPELINE_SOURCE == "merge_request_event"
   - if: $CI_COMMIT_REF_NAME == "master"
     variables:
      DESTINATION_FLAG: "--destination ${CI_REGISTRY_IMAGE}:latest"
   - if: $CI_COMMIT_TAG
     variables:
      DESTINATION_FLAG: "--destination ${CI_REGISTRY_IMAGE}:${CI_COMMIT_TAG}"
```

When on master (i.e. after a merge!), deploy the image with image tag “latest”

Build a docker image in gitlab

Can easily extend this to more rules:

When CI triggered by a git tag, deploy the image with the identical tag (e.g. “v1.0”)

```
image_build:
    [...]
    variables:
      DESTINATION_FLAG: "--no-push"
    rules:
      - if: $CI_PIPELINE_SOURCE == "merge_request_event"
      - if: $CI_COMMIT_REF_NAME == "master"
        variables:
          DESTINATION_FLAG: "--destination ${CI_REGISTRY_IMAGE}:latest"
      - if: $CI_COMMIT_TAG
        variables:
          DESTINATION_FLAG: "--destination ${CI_REGISTRY_IMAGE}:${CI_COMMIT_TAG}"
```