Automated testing - Preventing yourself and others from breaking your functioning code

Have you ever had some of these problems?:

- You change B and C, and suddenly A doesn't work anymore. Time wasted trying to figure out what changed.
- There was some simple problem, systematically testing could have found it.
- You get someone else's code and are afraid to touch it because who knows what might break. Plot twist: it's your own code!

People have learned that some automatic way to check problems makes software development much easier. This lesson will talk about the places it's useful for research code, and how easy it can be. We will discuss why testing often needs to be part of the software development cycle and how such a cycle can be implemented. We will see how automated testing works and practice designing and writing tests.

⚙ Prerequisites

1. You need pytest (as part of Anaconda or Miniconda or Virtual Environment).
2. (Optional) To work on exercises in other languages than Python, please follow the instructions under "Language-specific instructions" in the Test design episode to install the recommended testing frameworks.
3. Basic understanding of Git.
4. You need a GitHub or a Gitlab account for the "automated testing" and "full-cycle collaborative workflow" (but the rest works fine just locally).
5. If you wish to follow in the terminal and are new to the command line, we recorded a short shell crash course.

15 min	Motivation
25 min	Testing locally
30 min	Automated testing
30 min	Test design
5 min	Conclusions and recommendations

Motivation

Objectives

- Appreciate the importance of testing software
- Understand various benefits of testing

Untested software can be compared to uncalibrated detectors

"Before relying on a new experimental device, an experimental scientist always establishes its accuracy. A new detector is calibrated when the scientist observes its responses to known input signals. The results of this calibration are compared against the expected response."

[From Testing and Continuous Integration with Python, created by K. Huff]

With testing, simulations and analysis using software *can* be held to the same standards as experimental measurement devices!

What can go wrong when research software has bugs? Look no further:

- A Scientist's Nightmare: Software Problem Leads to Five Retractions
- Researchers find bug in Python script may have affected hundreds of studies

---

Testing in a nutshell

In software tests, expected results are compared with observed results in order to establish accuracy. Why are we not comparing directly all digits with the expected result?:

Python | C++ | R | Julia | Fortran

```python
def fahrenheit_to_celsius(temp_f):
    """Converts temperature in Fahrenheit
    to Celsius.
    """
    temp_c = (temp_f - 32.0) * (5.0/9.0)
    return temp_c

# This is the test function: `assert` raises an error if something
# is wrong.
def test_fahrenheit_to_celsius():
    temp_c = fahrenheit_to_celsius(temp_f=100.0)
    expected_result = 37.777777
    assert abs(temp_c - expected_result) < 1.0e-6
```

Or you can test whole programs:

```
$ python3 run-test.py

running: sample_data/set1.csv --output=tests/set1.txt
CORRECT
```

What can tests help you do?

Preserving expected functionality

- Check old things when you add new ones

Help users of your code

- Verify it's installed correctly and works.
- See examples of what it should do.

Help other developers modify it

- Change things with confidence that nothing is breaking.
- Warning if documentation/examples go out of date.

Manage complexity

- If code is easy to test, it's probably easier to maintain.
- The next lesson Modular code development demonstrates this.

Discussion: When is it OK not to add tests?

💬 Discussion: When is it OK not to add tests?

Vote in the notes and we'll discuss soon. **It is always a balance: there is no "always"/"never".**

1. Jupyter or R Markdown notebook which produces a plot and you know by looking at the plot whether it worked?
2. A short, "obviously correct" Python or R script which you never intend to reuse?
3. A simple short, "obviously correct" shell script?

4. Can you give other examples?

Discussion: What's easy and hard to test?

💬 Discussion: Testing in practice

Use the collaborative notes to answer these questions:

1. Give examples of things (from your work) that are easy to test.
2. Give examples of things (from your work) that are hard to test.

Types of tests

- Test functions one at a time - **Unit tests**
- Test how parts work together - **Integration tests**
- Test the whole thing running - **End-to-end tests**
  - For example, running on sample data.
- Check same results as before - **Regression tests**
- Write test first (the output), then write code to make test pass - **Test-driven development**
- GitHub or GitLab runs tests automatically - **Continuous integration**
- Report that tells you which lines were/were not run by tests - **Code coverage**
- Framework that runs test for you - **Testing framework**
  - See Quick Reference for some examples.

What should you do?

- Not every code needs perfect test coverage.
- If code is interactive-only (Jupyter Notebook), it's usually hard to test.
  - But also hard to run: the next lesson will discuss!
- At least end-to-end is often easy to add.
- Add tests of tricky functions.
  - If you'd have to run it over and over to test while writing, why not make it a property test?
- It's easy to have Gitlab/Github run the tests.
  - It's nice to push without thinking, and the system tells you when it's broke.
- **Learning how to test well make the rest of your code better, too.**

Where to start

- A simple script or notebook probably does not need an automated test.

If you have nothing yet

- Start with an end-to-end test.
- Describe in words how *you* check whether the code still works.
- Translate the words into a script.
- Run the script automatically on every code change.

If you want to start with unit-testing

- You want to rewrite a function? Start adding a unit test right there first.

Testing locally

❓ Questions

- How hard is it to set up a test suite for a first unit test?

Exercise

In this exercise we will make a simple function and use one of the language specific test frameworks to test it.

- This is easy to use by almost any project and doesn't rely on any other servers or services.
- The downside is that you have to remember to run it yourself.

✍️ Local-1: Create a minimal example (15 min)

In this exercise, we will create a minimal example using the pytest, run the test, and show what happens when a test breaks.

1. Create a new directory and change into it:

```
$ mkdir local-testing-example
$ cd local-testing-example
```

2. Create an example file and paste the following code into it

Python | R | Julia | C++

Create `example.py` with content

```python
def add(a, b):
    return a + b

def test_add():
    assert add(2, 3) == 5
    assert add('space', 'ship') == 'spaceship'
```

This code contains one genuine function and a test function. `pytest` finds any functions beginning with `test_` and treats them as tests.

3. Run the test

Python | R | Julia | C++

```
$ pytest -v example.py

============================================================ test session
starts ================================
platform linux -- Python 3.7.2, pytest-4.3.1, py-1.8.0, pluggy-0.9.0 --
/home/user/pytest-example/venv/bin/python3
cachedir: .pytest_cache
rootdir: /home/user/pytest-example, inifile:
collected 1 item

example.py::test_add PASSED

============================================================ 1 passed in 0.01
seconds ===============================
```

Yay! The test passed!

Hint for participants trying this inside Spyder or IPython: try `!pytest -v example.py`.

4. Let us break the test!

Introduce a code change which breaks the code (e.g. `-` instead of `+`) and check whether our test detects the change:

Python | R | Julia | C++

```
$ pytest -v example.py

============================================================= test session
starts ==============================
platform linux -- Python 3.7.2, pytest-4.3.1, py-1.8.0, pluggy-0.9.0 --
/home/user/pytest-example/venv/bin/python3
cachedir: .pytest_cache
rootdir: /home/user/pytest-example, inifile:
collected 1 item

example.py::test_add FAILED

================================================================= FAILURES
=================================
_____________________________________________________________________ test_add
_____________________________________

    def test_add():
>       assert add(2, 3) == 5
E       assert -1 == 5
E         --1
E         +5

example.py:6: AssertionError
========================================================== 1 failed in 0.05
seconds ==============
```

Notice how pytest is smart and includes context: lines that failed, values of the relevant variables.

✍️ (optional) Local-2: Create a test that considers numerical tolerance (10 min)

Let's see an example where the test has to be more clever in order to avoid false negative.

In the above exercise we have compared integers. In this optional exercise we want to learn how to compare floating point numbers since they are more tricky (see also "What Every Programmer Should Know About Floating-Point Arithmetic").

The following test will fail and this might be surprising. Try it out:

Python | R | Julia | C++

```python
def add(a, b):
    return a + b

def test_add():
    assert add(0.1, 0.2) == 0.3
```

Your goal: find a more robust way to test this addition.

✔ Solution: Local-2

Python | R | Julia | C++

One solution is to use pytest.approx:

```
from pytest import approx

def add(a, b):
    return a + b

def test_add():
    assert add(0.1, 0.2) == approx(0.3)
```

But maybe you didn't know about pytest.approx: and did this instead:

```
def test_add():
    result = add(0.1, 0.2)
    assert abs(result - 0.3) < 1.0e-7
```

This is OK but the `1.0e-7` can be a bit arbitrary.

❗ Keypoints

- Each test framework has its way of collecting and running all test functions, e.g. functions beginning with `test_` for `pytest`.
- Python, Julia and C/C++ have better tooling for automated tests than Fortran and you can use those also for Fortran projects (via `iso_c_binding`).

Automated testing

❓ Questions

- How can we implement automatic testing each time we push changes to the repository?

- Why is it good to autoclose issues with commit messages?

Continuous integration

We will now learn to set up automatic tests using either GitHub Actions or GitLab CI - you can choose which one to use and instructions are provided for both.

This exercise can be run in "collaborative mode" by following instead the instructions in Full-cycle collaborative workflow. In the collaborative version steps C-D below are performed by a collaborator.

✍ Exercise CI-1: Create and use a continuous integration workflow on GitHub or GitLab

In this exercise, we will:

- **A.** Create and add code to a repository on GitHub/GitLab (or, alternatively, fork and clone an existing example repository)
- **B.** Set up tests with GitHub Actions/ GitLab CI
- **C.** Find a bug in our repository and open an issue to report it
- **D.** Fix the bug on a bugfix branch and open a pull request (GitHub)/ merge request (GitLab)
- **E.** Merge the pull/merge request and see how the issue is automatically closed.
- **F.** Create a test to increase the code coverage of our tests.

Prerequisites

If you are new to Git, you can find a step-by-step guide to setting up repositories and making commits in this git-refresher material. If you are new to pull requests / merge requests, you can learn all about them in the Collaborative Git lesson.

Step 1: Create a new repository on GitHub/GitLab OR fork from the example repo

Create a new repository

- Begin by creating a repository called (for example) *example-ci*.
- **Before** you create the repository, select **"Initialize this repository with a README"** (otherwise you try to clone an empty repo).
- Clone the repository (`git clone git@github.com:<yourGitID>/example-ci.git`).
- Add the following files and code

Python | R

Add a file `functions.py` containing:

```python
def add(a, b):
    return a + b

def subtract(a, b):
    return a + b  # <--- fix this in step 7

def multiply(a, b):
    return a * b

def convert_fahrenheit_to_celsius(fahrenheit):
    return multiply(subtract(fahrenheit, 32), 9 / 5) # <-- Fix this in step 7
```

and a file `test_functions.py` containing:

```python
from functions import add, subtract, multiply
from functions import convert_fahrenheit_to_celsius as f2c
import pytest

def test_add():
    assert add(2, 3) == 5
    assert add('space', 'ship') == 'spaceship'

# uncomment the following test in step 5
#def test_subtract():
#    assert subtract(2, 3) == -1

# uncomment the following test in step 11
# def test_convert_fahrenheit_to_celsius():
#    assert f2c(32) == 0
#    assert f2c(122) == pytest.approx(50)
#    with pytest.raises(AssertionError):
#        f2c(-600)
```

Finally, stage the files (`git add <filename>`), commit (`git commit -m "some commit message"`), and push the changes (`git push origin main`).

Fork and clone an existing example repository

- Fork the example repo. There are two options one for Python and one for R.
- Clone your fork (`git clone git@github.com:<yourGitID>/<Py/R>TestingExample.git`).

Step 2: Run tests locally

Python | R

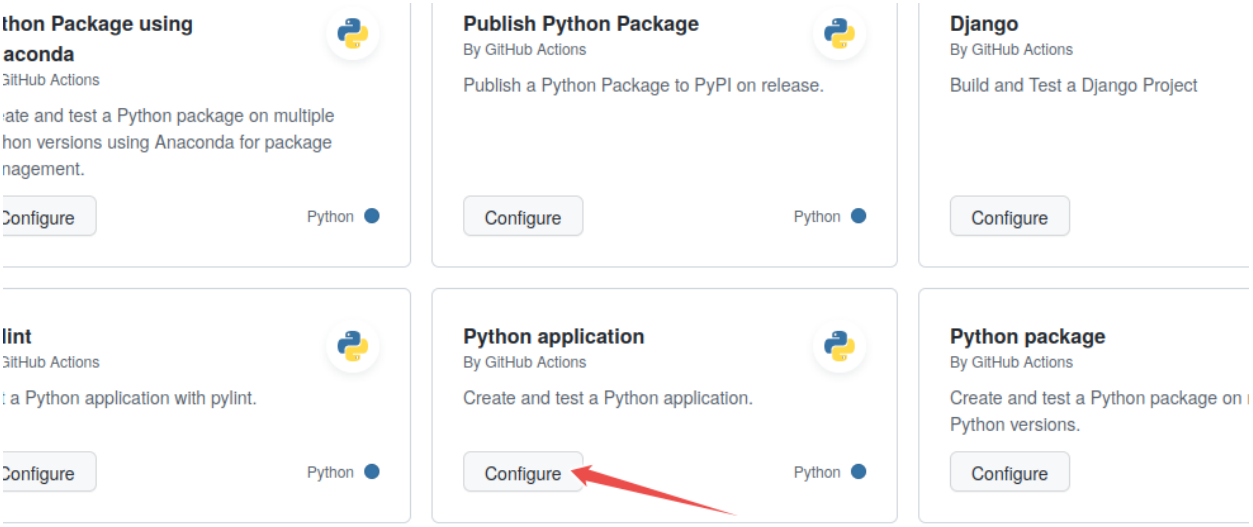
You can now run your tests locally with

```
pytest
```

Step 3: Enable automated testing

GitHub-Python | GitLab-Python | GitHub-R

In this step we will enable GitHub Actions. Select “Actions” from your GitHub repository page. You get to a page “Get started with GitHub Actions”. Select the button for “Configure” under Python Application:

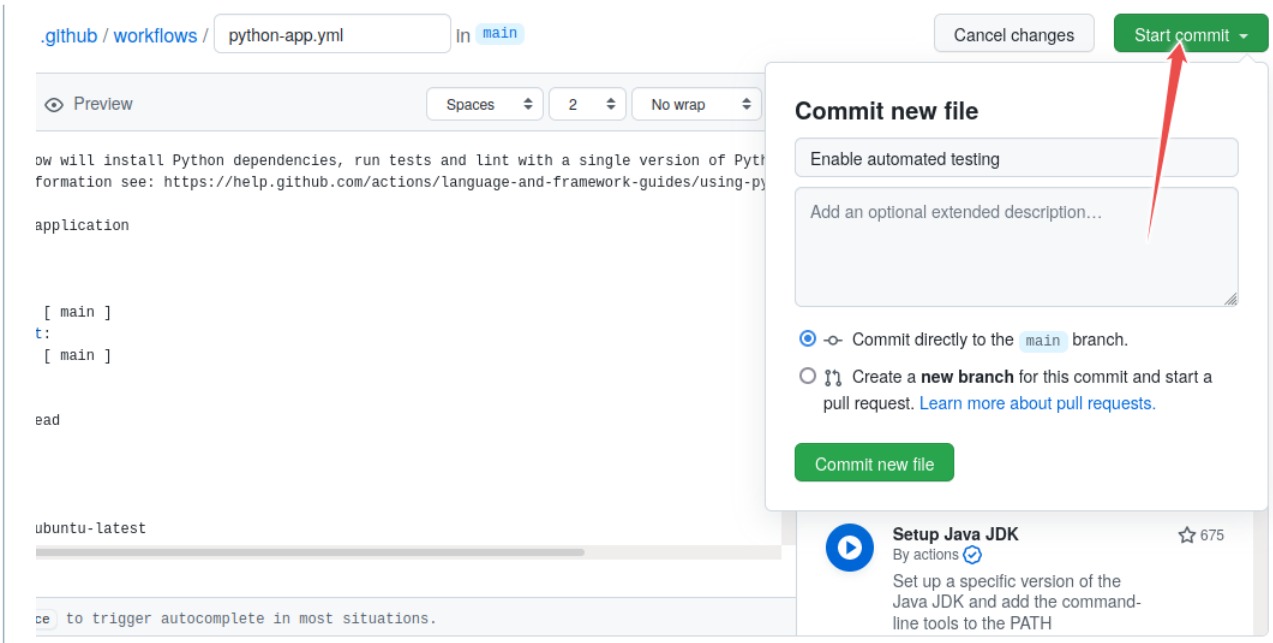


Select “Python application” as the starter workflow.

GitHub creates the following file for you in the subfolder `.github/workflows`. Modify the highlighted lines according to the action below. This will add a code coverage report to new pull requests. The if clause restricts this to pull requests, as otherwise this action would not have a target to write the reports to. On pushes only the unittesting is run.

```yaml
# This workflow will install Python dependencies, run tests and lint with a single
version of Python
# For more information see: https://docs.github.com/en/actions/automating-builds-
and-tests/building-and-testing-python

name: Test

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

permissions:
  contents: read
  pull-requests: write

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v4
    - name: Set up Python 3.10
      uses: actions/setup-python@v3
      with:
        python-version: "3.10"
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install flake8 pytest pytest-cov
        if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
    - name: Lint with flake8
      run: |
        # stop the build if there are Python syntax errors or undefined names
        flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
        # exit-zero treats all errors as warnings. The GitHub editor is 127 chars
wide
        flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 --
statistics
    - name: Test with pytest
      run: |
        pytest --cov-report "xml:coverage.xml" --cov=.
    - name: Create Coverage
      if: ${{ github.event_name == 'pull_request' }}
      uses: orgoro/coverage@v3
      with:
          coverageFile: coverage.xml
          token: ${{ secrets.GITHUB_TOKEN }}
```

Commit the change by pressing the “Start Commit” button:

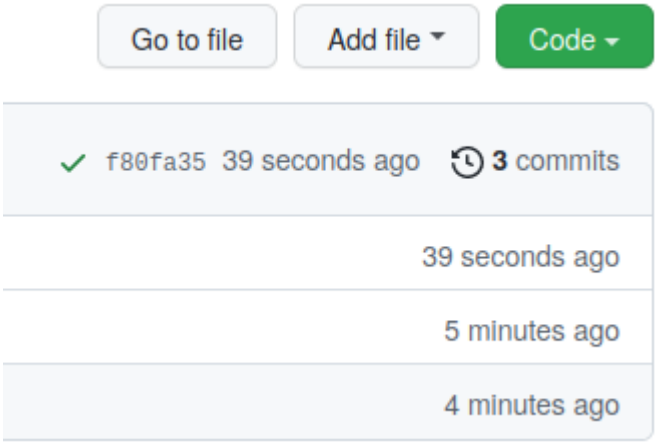


Committing the file via the GitHub web interface: follow the flow, give it some commit name. You can commit directly to master.

Step 4: Verify that tests have been automatically run

GitHub-Python | GitLab-Python | GitHub-R

Observe in the repository how the test succeeds. While the test is executing, the repository has a yellow marker. This is replaced with a green check mark, once the test succeeds:



Green check means passed.

Also browse the “Actions” tab and look at the steps there and their output.

Step 5: Add a test which reveals a problem

After you committed the workflow file, your GitHub/GitLab repository will be ahead of your local cloned repository. Update your local cloned repository:

```
$ git pull origin main
```

Hint: if the above command fails, check whether the branch name on the GitHub/GitLab repository is called `main` and not perhaps `master`.

Next uncomment the code in `test_functions.py` under "step 5", commit, and push. Verify that the test suite now fails on the "Actions" tab (GitHub) or the "CI/CD->Pipelines" tab (GitLab).

Step 6: Open an issue on GitHub/GitLab

Open a new issue in your repository about the broken test (click the "Issues" button on GitHub or GitLab and write a title for the issue). The plan is that we will fix the issue through a pull/merge request.

Step 7: Fix the broken test

Now fix the code **on a new branch**, you can call it `yourname/bugfix`. After you have fixed the code on the new branch, commit the following commit message `"restore function subtract; fixes #1"` (assuming that you try to fix issue number 1).

Shortcut

Here it's perfectly possible to take a shortcut and commit and push directly to the main branch. If you do this, steps 8-9 below are skipped.

- When would you push directly to the main branch, and when would you send a pull/merge request?

Then push to your repository.

Step 8: Open a pull request (GitHub)/ merge request (GitLab)

Go back to the repository on GitHub or GitLab and open a pull/merge request. **In a collaborative setting, you could request a code review from collaborators at this stage.** Before accepting the pull/merge request, observe how GitHub Actions/ Gitlab CI automatically tested the code.

If you forgot to reference the issue number in the commit message, you can still add it to the pull/merge request: `my pull/merge request title, closes #1`.

Step 9: Accept the pull/merge request

Observe how accepting the pull/merge request automatically closes the issue (provided the commit message or the pull/merge request contained the correct issue number).

See also:

- GitHub: closing issues using keywords
- GitLab: closing issues using keywords

Discuss whether this is a useful feature. And if it is, why do you think is it useful?

Step 10: Increase your code coverage

We are currently missing several functions in our tests. Write a test for the `multiply` function in a new branch and create a pull request. On Python you can directly observe the increase in code coverage. On R you can have a look at the action (`Actions -> last run of your action -> Select a job -> Test coverage`). If you compare this with the previous run, you should see an increase once the update is in.

Step 11 (optional): Repeat steps 5-9 for the `convert_fahrenheit_to_celsius` function:

Repetition helps learning, so let's do the testing again for our `convert_fahrenheit_to_celsius` function. Uncomment the test for the `convert_fahrenheit_to_celsius` function and repeat steps 5 to 9 fixing the bug this test exposes.

Discussion

Finally, we discuss together about our experiences with this exercise.

Where to go from here

- This example was using Python but you can achieve the same automation for R or Fortran or C/C++ or other languages
- This workflow is very useful for collaborators who work on the same code and it works both for centralized and forking workflows - have a look at this alternative exercise to see how that works.
- GitHub Actions has a Marketplace which offer wide range of automatic workflows
- On GitLab use GitLab CI
- For Windows builds you can also use Appveyor

Keypoints

- When fixing bugs or other problems reported in issues, use the issue autoclosing mechanism when you send the pull/merge request.

Test design

❓ Questions

- How can different types of functions and classes be tested?
- How can the integrity of a complete program be monitored over time?
- How can functions that involve random numbers be tested?

In this episode we will consider how functions and programs can be tested in programs developed in different programming languages.

💬 Exercise instructions

For the instructor

- First motivate and give a quick tour of all exercises below (10 minutes).
- Emphasize that the focus of this episode is *design*. It is OK to only discuss in groups and not write code.

During exercise session

- Choose the exercise which interests you most. There are many more exercises than we would have time for.
- Discuss what testing framework can be used to implement the test.
- Keep notes, questions, and answers in the collaborative document.

Once we return to stream

- Discussion about experiences learned.

❗ Language-specific instructions

Python | C++ | R | Julia | Fortran

The suggested solutions below use pytest. Further information can be found in the Quick Reference.

Pure and impure functions

Start by discussing how you would design tests for the following five functions, and then try to write the tests. Also discuss why some are easier to test than others.

✍ Design-1: Design a test for a function that receives a number and returns a number

Python | C++ | R | Julia | Fortran

```python
def factorial(n):
    """
    Computes the factorial of n.
    """
    if n < 0:
        raise ValueError('received negative input')
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

Discussion point: The factorial grows very rapidly. What happens if you pass a large number as argument to the function?

✔ Solution

This is a **pure function** so is easy to test: inputs go to outputs. For example, start with the below, then think of some what extreme cases/boundary cases there might be. This example shows all of the tests as one function, but you might want to make each test function more fine-grained and test only one concept.

Python | C++ | R | Julia | Fortran

```python
import pytest

def test_factorial():
    assert factorial(0) == 1
    assert factorial(1) == 1
    assert factorial(2) == 2
```

Notes on the discussion point: Programming languages differ in the way they deal with integer overflow. Python automatically converts to the necessary `long` type, in Julia you would observe a "wrap-around", in C/C++ you get undefined behaviour for signed integers. Testing for overflow likewise depends on the language.

✍ Design-2: Design a test for a function that receives two strings and returns a number

Python | C++ | R | Julia | Fortran

```python
def count_word_occurrence_in_string(text, word):
    """
    Counts how often word appears in text.
    Example: if text is "one two one two three four"
             and word is "one", then this function returns 2
    """
    words = text.split()
    return words.count(word)
```

✔ Solution

This is again a **pure function** but uses strings. Use a similar strategy to the above.

Python | C++ | R | Julia | Fortran

```python
def test_count_word_occurrence_in_string():
    assert count_word_occurrence_in_string('AAA BBB', 'AAA') == 1
    assert count_word_occurrence_in_string('AAA AAA', 'AAA') == 2
    # What does this last test tell us?
    assert count_word_occurrence_in_string('AAAAA', 'AAA') == 1
```

✍ Design-3: Design a test for a function which reads a file and returns a number

Python | C++ | R | Julia | Fortran

```python
def count_word_occurrence_in_file(file_name, word):
    """
    Counts how often word appears in file file_name.
    Example: if file contains "one two one two three four"
             and word is "one", then this function returns 2
    """
    count = 0
    with open(file_name, 'r') as f:
        for line in f:
            words = line.split()
            count += words.count(word)
    return count
```

✔ Solution

In this example we test a function which is not pure, because the output depends on the value of a file. We can generate a temporary file for testing and remove it afterwards. Even better could be to split file reading from the calculation, so that testing the calculation part becomes easy (see above).

Python | C++ | R | Julia | Fortran

```python
import tempfile
import os

def test_count_word_occurrence_in_file():
    _, temporary_file_name = tempfile.mkstemp()
    with open(temporary_file_name, 'w') as f:
        f.write("one two one two three four")
    count = count_word_occurrence_in_file(temporary_file_name, "one")
    assert count == 2
    os.remove(temporary_file_name)
```

✍ Design-4: Design a test for a function with an external dependency

This one is not easy to test because the function has an external dependency.

Python | C++ | R | Julia | Fortran

```python
def check_reactor_temperature(temperature_celsius):
    """
    Checks whether temperature is above max_temperature
    and returns a status.
    """
    from reactor import max_temperature
    if temperature_celsius > max_temperature:
        status = 1
    else:
        status = 0
    return status
```

✔ Solution

This function depends on the value of `reactor.max_temperature` so the function is not pure, so testing gets harder. You could use monkey patching to override the value of `max_temperature`, and test it with different values. Monkey patching is the concept of artificially changing some other value.

A better solution would probably be to rewrite the function.

Python | C++ | R | Julia | Fortran

```python
def test_set_temp(monkeypatch):
    monkeypatch.setattr(reactor, "max_temperature", 100)
    assert check_reactor_temperature(99)  == 0
    assert check_reactor_temperature(100) == 0   # boundary cases easily go
wrong
    assert check_reactor_temperature(101) == 1
```

✍ Design-5: Design a test for a method of a mutable class

Python | C++ | R | Julia | Fortran

```python
class Pet:
    def __init__(self, name):
        self.name = name
        self.hunger = 0
    def go_for_a_walk(self):  # <-- how would you test this function?
        self.hunger += 1
```

✔ Solution

Python C++ R Julia Fortran

```
def test_pet():
    p = Pet('fido')
    assert p.hunger == 0
    p.go_for_a_walk()
    assert p.hunger == 1

    p.hunger = -1
    p.go_for_a_walk()
    assert p.hunger == 0
```

Test-driven development

✍ Design-6: Experience test-driven development

Write a test before writing the function! You can decide yourself what your unwritten function should do, but as a suggestion it can be based on FizzBuzz - i.e. a function that:

- takes an integer argument
- for arguments that are multiples of three, returns “Fizz”
- for arguments that are multiples of five, returns “Buzz”
- for arguments that are multiples of both three and five, returns “FizzBuzz”
- fails in case of non-integer arguments or integer arguments 0 or negative
- otherwise returns the integer itself

When writing the tests, consider the different ways that the function could and should fail.

After you have written the tests, implement the function and run the tests until they pass.

✔ Solution

Python C++ R Julia Fortran

```
import pytest

def fizzbuzz(number):
    if not isinstance(number, int):
        raise TypeError
    if number < 1:
        raise ValueError
    elif number % 15 == 0:
        return "FizzBuzz"
    elif number % 3 == 0:
        return "Fizz"
    elif number % 5 == 0:
        return "Buzz"
    else:
        return number

def test_fizzbuzz():
    expected_result = [1, 2, "Fizz", 4, "Buzz", "Fizz",
                       7, 8, "Fizz", "Buzz", 11, "Fizz",
                       13, 14, "FizzBuzz", 16, 17, "Fizz", 19, "Buzz"]
    obtained_result = [fizzbuzz(i) for i in range(1, 21)]

    assert obtained_result == expected_result

    with pytest.raises(ValueError):
        fizzbuzz(-5)
    with pytest.raises(ValueError):
        fizzbuzz(0)

    with pytest.raises(TypeError):
        fizzbuzz(1.5)
    with pytest.raises(TypeError):
        fizzbuzz("rabbit")

def main():
    for i in range(1, 100):
        print(fizzbuzz(i))

if __name__ == "__main__":
    main()
```

Testing randomness

How would you test functions which generate random numbers according to some distribution/statistics?

Functions and modules which contain randomness are more difficult to test than pure deterministic functions, but many strategies exist:

- For unit tests we can use fixed random seeds.
- Try to test whether your results follow the expected distribution/statistics.

- When you verify your code “by eye”, what are you looking at? Now try to express that in a script.

✍️ Design-7: Write two different types of tests for randomness

Consider the code below which simulates playing Yahtzee by using random numbers. How would you go about testing it?

Try to write two types of tests:

- a *unit test* for the `roll_dice` function. Since it uses random numbers, you will need to **set the random seed**, pre-calculate what sequence of dice throws you get with that seed, and use that in your test.
- a test of the `yahtzee` function which considers the statistical probability of obtaining a “Yahtzee” (5 dice with the same value after three throws), which is around 4.6%. This test will be an *integration test* since it tests multiple functions including the random number generator itself.

Python | C++ | R | Julia

```python
import random
from collections import Counter


def roll_dice(num_dice):
    return [random.choice([1, 2, 3, 4, 5, 6]) for _ in range(num_dice)]


def yahtzee():
    """
    Play yahtzee with 5 6-sided dice and 3 throws.
    Collect as many of the same dice side as possible.
    Returns the number of same sides.
    """

    # first throw
    result = roll_dice(5)
    most_common_side, how_often = Counter(result).most_common(1)[0]

    # we keep the most common side
    target_side = most_common_side
    num_same_sides = how_often
    if num_same_sides == 5:
        return 5

    # second and third throw
    for _ in [2, 3]:
        throw = roll_dice(5 - num_same_sides)
        num_same_sides += Counter(throw)[target_side]
        if num_same_sides == 5:
            return 5

    return num_same_sides


if __name__ == "__main__":
    num_games = 100

    winning_games = list(
        filter(
            lambda x: x == 5,
            [yahtzee() for _ in range(num_games)],
        )
    )

    print(f"out of the {num_games} games, {len(winning_games)} got a yahtzee!")
```

✔ Solution

Python | C++ | R | Julia

```python
def test_roll_dice():
    random.seed(0)
    assert roll_dice(5) == [4, 4, 1, 3, 5]
    assert roll_dice(5) == [4, 4, 3, 4, 3]
    assert roll_dice(5) == [5, 2, 5, 2, 3]


import pytest
def test_yahtzee():
    random.seed(1)
    n_tests = 1_000_000

    winning_games = list(
        filter(
            lambda x: x == 5,
            [yahtzee() for _ in range(n_tests)],
        )
    )

    assert len(winning_games) / n_tests == pytest.approx(0.046, abs=0.01)
```

Designing an end-to-end test

In this exercise we will practice designing an end-to-end test. In an end-to-end test (or integration test), the unit is the entire program. We typically feed the program with some well defined input and verify that it still produces the expected output by comparing it to some reference.

✍ Design-8: Design (but not write) an end-to-end test for the uniq program

To have a tangible example, let us consider the `uniq` command. This command can read a file or an input stream and remove consecutive repetition. The program behind `uniq` has been written by somebody else, it probably contains some functions, but we will not look into it but regard it as “black box”.

If we have a file called `repetitive-text.txt` containing:

```
(all together now) all together now
(all together now) all together now
(all together now) all together now
(all together now) all together now
(all together now) all together now
another line
another line
another line
another line
intermission
more repetition
more repetition
more repetition
more repetition
more repetition
(all together now) all together now
(all together now) all together now
```

... then feeding this input file to `uniq` like this:

```
$ uniq < repetitive-text.txt
```

... will produce the following output with repetitions removed:

```
(all together now) all together now
another line
intermission
more repetition
(all together now) all together now
```

How would you write an end-to-end test for `uniq`?

✍ Design-9: More end-to-end testing

- Now imagine a code which reads numbers and produces some (floating point) numbers. How would you test that?
- How would you test a code end-to-end which produces images?

✍ Design-10: Create an actual end-to-end test

Often, you can include tests that run your whole workflow or program. For example, you might include sample data and check the output against what you expect. (including sample data is a great idea anyway, so this helps a lot!)

We'll use the word-count example repository https://github.com/coderefinery/word-count.

As a reminder, you can run the script like this to get some output, which prints to standard output (the terminal):

```
$ python3 code/count.py data/abyss.txt
```

Your goal is to make a test that can run this and let you know if it's successful or not. You could use Python, or you could use shell scripting. You can test if these two lines are in the output: `the 4044` and `and 2807`.

Python hint: subprocess.check_output will run a command and return its output as a string.

Bash hint: `COMMAND | grep "PATTERN"` ("pipe to grep") will be true if the pattern is in the command.

✔ Solution

There are two solutions in the repository already, in the `tests/` dierectory https://github.com/coderefinery/word-count, one in Python and one in bash shell. Neither of these are a very advanced or perfect solution, and you could integrate them with pytest or whatever other test framework you use.

The shell one works with shell and prints a bit more output:

```
$ sh tests/end-to-end-shell.sh
the 4044
Success: 'the' found correct number of times
and 2807
Success: 'and' found correct number of times
Success
```

The Python one:

```
$ python3 tests/end-to-end-python.py
Success
```

Keypoints

- Pure functions (these are functions without side-effects) are easiest to test. And also easiest to reuse in another code project since they don't depend on any side-effects.
- Classes can be tested but it's somewhat more elaborate.

Conclusions and recommendations

The basics

- Learn one test framework well enough for basics
  - Explore and use the good tools that exist out there
  - An incomplete list of testing frameworks can be found in the Quick Reference
- Start with some basics
  - Some simple thing that test all parts
- Automate tests
  - Faster feedback and reduce the number of surprises

Going more in-depth

- Strike a healthy balance between unit tests and integration tests
- As the code gets larger and the chance of undetected bugs increases, tests should increase
- When adding new functionality, also add tests
- When you discover and fix a bug, also commit a test against this bug
- Use code coverage analysis to identify untested or unused code
- If you make your code easier to test, it becomes more modular

Ways to get started

You probably won't do everything perfectly when you start off... But what are some of the easy starting points?

- Do you have some single functions that are easy to test, but hard to verify just by looking at them? Add unit tests.
- Do you have data analysis or simulation of some sort? Make an end-to-end test with sample data, or sample parameters. This is useful as an example anyway.
- A local testing framework + GitHub actions is very easy! And works well in the background - you do whatever you want and get an email if you break things. It's actually pretty freeing.

(Optional) Full-cycle collaborative workflow

❓ Questions

- How can we implement automatic testing each time we push changes to the repository?

- Why is it good to autoclose issues with commit messages?

Exercise a full-cycle collaborative workflow

This exercise is a collaborative version of the Automated testing exercise.

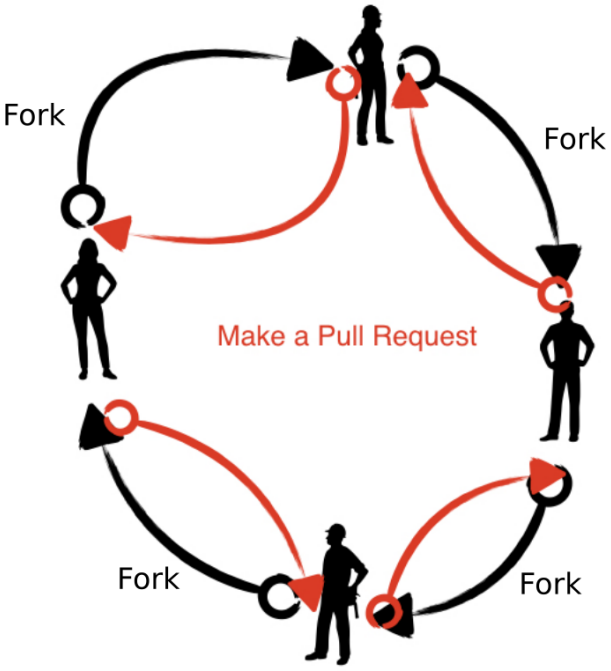
✍ FullCI-1: Create and use a continuous integration workflow on GitHub or GitLab with pull requests and issues

This is an expanded version of the automated testing demonstration. The exercise is performed in a collaborative circle within the exercise group (breakout room).

The exercise takes 20-30 minutes.

In this exercise, everybody will:

A. Create a repository on GitHub/GitLab (everybody should use a **different repository name** for their repository) **B.** Commit code to the repository and set up tests with GitHub Actions/ GitLab CI **C.** Everybody will find a bug in their repository and open an issue in their repository **D.** Then each one will clone the repo of one of their exercise partners, fix the bug, and open a pull request (GitHub)/ merge request (GitLab) **E.** Everybody then merges their co-worker's change



Overview of this exercise. Below we detail the steps.

Prerequisites

If you are new to Git, you can find a step-by-step guide to setting up repositories and making commits in this git-refresher material.

Step 1: Create a new repository on GitHub/GitLab

- Select a **different repository name** than your colleagues (otherwise forking the same name will be strange)
- **Before** you create the repository, select **"Initialize this repository with a README"** (otherwise you try to clone an empty repo).
- Share the repository URL with your exercise group via shared document or chat

Step 2: Clone your own repository, add code, commit, and push

Clone the repository.

Add a file `example.py` containing:

```
def add(a, b):
    return a + b


def test_add():
    assert add(2, 3) == 5
    assert add('space', 'ship') == 'spaceship'


def subtract(a, b):
    return a + b  # <--- fix this in step 8


# uncomment the following test in step 5
#def test_subtract():
#    assert subtract(2, 3) == -1
```

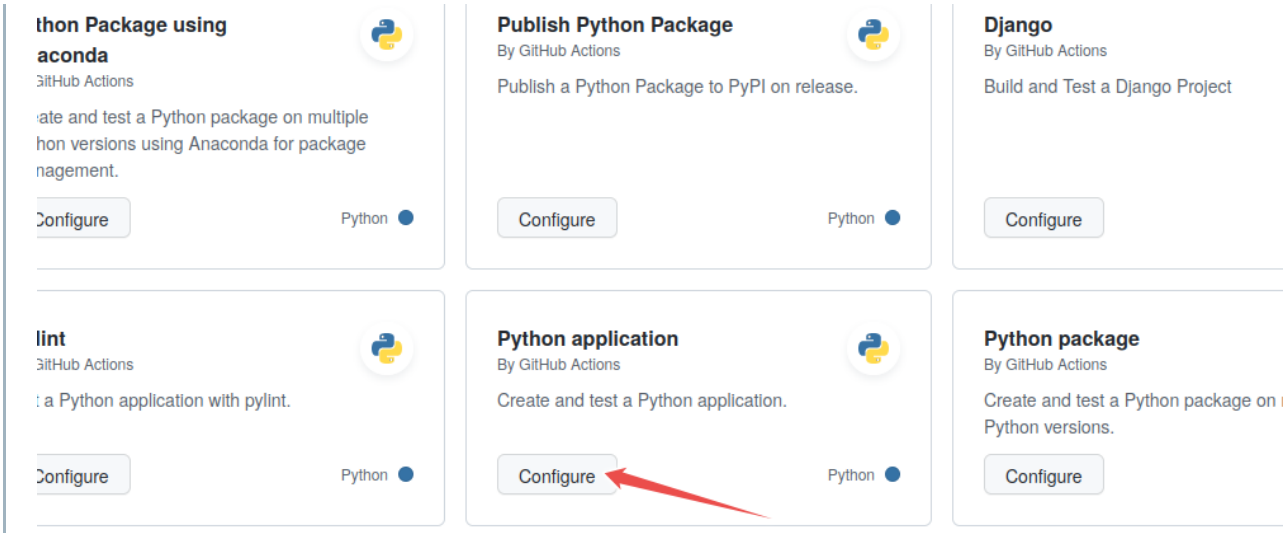
Test `example.py` with `pytest`.

Then stage the file (`git add <filename>`), commit (`git commit -m "some commit message"`), and push the changes (`git push origin main`).

Step 3: Enable automated testing

GitHub | GitLab

In this step we will enable GitHub Actions. Select "Actions" from your GitHub repository page. You get to a page "Get started with GitHub Actions". Select the button for "Set up this workflow" under Python Application:

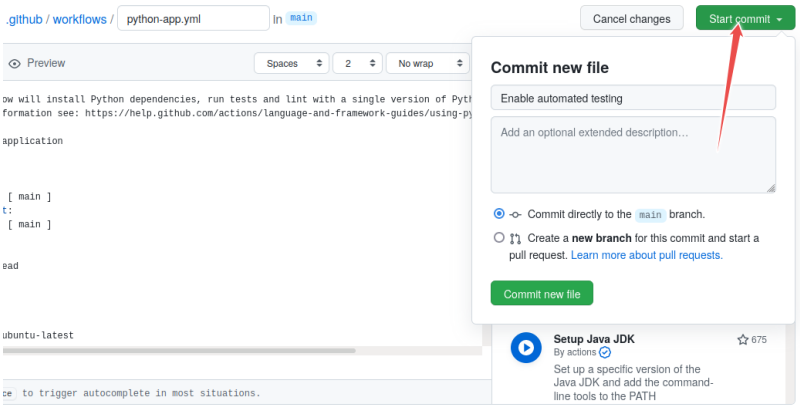


Select "Python application" as the starter workflow.

GitHub creates the following file for you in the subfolder `.github/workflows`. Add `pytest example.py` to the last line (highlighted):

```yaml
# This workflow will install Python dependencies, run tests and lint with a single version of Python
# For more information see: https://help.github.com/actions/language-and-framework-guides/using-python-with-github-actions

name: Python application

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - name: Set up Python 3.8
      uses: actions/setup-python@v2
      with:
        python-version: 3.8
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install flake8 pytest
        if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
    - name: Lint with flake8
      run: |
        # stop the build if there are Python syntax errors or undefined names
        flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
        # exit-zero treats all errors as warnings. The GitHub editor is 127 chars wide
        flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 --statistics
    - name: Test with pytest
      run: |
        pytest example.py
```

Commit the change by pressing the “Start Commit” button:

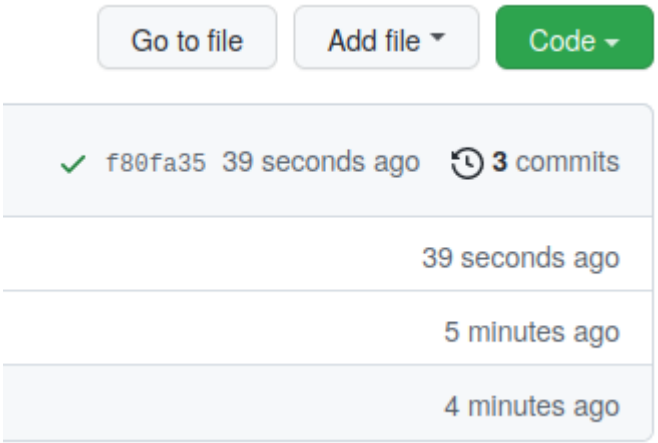


Committing the file via the GitHub web interface: follow the flow, give it some commit name. You can commit directly to master.

Step 4: Verify that tests have been automatically run

GitHub | GitLab

Observe in the repository how the test succeeds. While the test is executing, the repository has a yellow marker. This is replaced with a green check mark, once the test succeeds:



Green check means passed.

Also browse the “Actions” tab and look at the steps there and their output.

Step 5: Add a test which reveals a problem

After you committed the workflow file, your GitHub/GitLab repository will be ahead of your local cloned repository. Update your local cloned repository:

```
$ git pull origin main
```

Hint for helpers: if the above command fails, check whether the branch name on the GitHub/GitLab repository is called `main` and not perhaps `master`.

Next uncomment the code in `example.py` under "step 5", commit, and push. Verify that the test suite now fails on the "Actions" tab (GitHub) or the "CI/CD->Pipelines" tab (GitLab).

Step 6: Open an issue on GitHub/GitLab

Open a new issue in your repository about the broken test (click the "Issues" button on GitHub or GitLab and write a title for the issue). The plan is that your colleague will fix the issue through a pull/merge request.

Step 7: Fork and clone the repository of your colleague

Fork the repository using the GitHub/GitLab web interface. (if you are unfamiliar with forking and pull requests, have a look at this visual representation).

Make sure you clone the fork after you have forked it. Do not clone your colleague's repository directly.

GitHub | GitLab

```
$ git clone https://github.com/your-username/the-repository.git
```

Step 8: Fix the broken test

After you have fixed the code, commit the following commit message `"restore function subtract; fixes #1"` (assuming that you try to fix issue number 1).

Then push to your fork.

Step 9: Open a pull request (GitHub)/ merge request (GitLab)

Then before accepting the pull request/ merge request from your colleague, observe how GitHub Actions/ Gitlab CI automatically tested the code.

If you forgot to reference the issue number in the commit message, you can still add it to the pull request/ merge request: `my pull/merge request title, closes #NUMBEROFTHEISSUE`

Step 10: Accept the pull request/ merge request

Observe how accepting the pull request/ merge request automatically closes the issue (provided the commit message or the pull request/ merge request contained the correct issue number).

See also:

- GitHub: closing issues using keywords
- GitLab: closing issues using keywords

Discuss whether this is a useful feature. And if it is, why do you think is it useful?

Discussion

Finally, we discuss together about our experiences with this exercise.

✍ (optional) FullCI-2: Add a license file to the previous exercise's repository

In the Social coding and open software lesson we learn how important it is to add a LICENSE file.

Your goal:

- You discover that your coworker's repository does not have a LICENSE file.
- Open an issue and suggest a LICENSE.
- Then add a LICENSE via a pull/merge request, referencing the issue number.

Where to go from here

- This example was using Python but you can achieve the same automation for R or Fortran or C/C++ or other languages
- GitHub Actions has a Marketpace which offer wide range of automatic workflows
- On GitLab use GitLab CI
- For Windows builds you can also use Appveyor

Keypoints

- When fixing bugs or other problems reported in issues, use the issue autoclosing mechanism when you send the pull/merge request.

List of exercises

Full list

This is a list of all exercises and solutions in this lesson, mainly as a reference for helpers and instructors. This list is automatically generated from all of the other pages in the lesson. Any single teaching event will probably cover only a subset of these, depending on their interests.

Quick Reference

Available tools

Unit test frameworks

A **test framework** makes it easy to run tests across large amounts of code automatically. They provide more control than one single script which does some tests.

- Python
  - pytest
  - doctest
  - unittest
- R
  - testthat
- Julia
  - Test
- Matlab
  - Class-Based Unit Tests
- C(++)
  - Google Test
  - Catch2
  - cppunit
  - Boost.Test
- Fortran
  - pFUnit
  - FRUIT
  - Ftunit

---

pytest

- Python
- https://docs.pytest.org/
- Installable via Conda or pip.

- Easy to use: Prefix a function with `test_` and the test runner will execute it. No need to subclass anything.

```
def get_word_lengths(s):
    """
    Returns a list of integers representing
    the word lengths in string s.
    """
    return [len(w) for w in s.split()]

def test_get_word_lengths():
    text = "Three tomatoes are walking down the street"
    assert get_word_lengths(text) == [5, 8, 3, 7, 4, 3, 6]
```

- Example project

---

testthat

- R
- https://github.com/r-lib/testthat
- Easily installed from CRAN with `install.packages("testthat")`, or from GitHub with `devtools::install_github("r-lib/testthat")`
- Use in package development with `usethis::use_testthat()`
- Add a new test file with `usethis::use_test("test-name")`, e.g.:

```
# tests/testthat/test_example.R
# file added by running `usethis::use_test("example")`

context("Arithmetics")
library("mypackage")

test_that("square root function works", {
  expect_equal(my_sqrt(4), 2)
  expect_warning(my_sqrt(-4))
})
```

  Tests consist of one or more *expectations*, and multiple tests can be grouped together in one test file. Test files are put in the directory `tests/testthat/`, and their file names are prefixed with `test_`.

- Run all tests in package with `devtools::test()` (if you use RStudio, press `Ctrl+Shift+T`):

```
> devtools::test()
Loading mypackage
Testing mypackage
✔ |  OK F W S | Context
✔ |   2       | Arithmetics

══ Results ═══════════════════════════════════════════════════════════════════
OK:       2
Failed:   0
Warnings: 0
Skipped:  0
```

More information in the Testing chapter of the book R Packages by Hadley Wickham.

Test

- Julia
- - Documentation
- Part of the standard library
- Provides simple unit testing functionality with `@test` and `@test_throws` macros:

```
julia> using Test

julia> @test [1, 2] + [2, 1] == [3, 3]
Test Passed

# approximate comparisons:
julia> @test π ≈ 3.14 atol=0.01
Test Passed

# Tests that an expression throws exception:
julia> @test_throws BoundsError [1, 2, 3][4]
Test Passed
      Thrown: BoundsError

julia> @test_throws DimensionMismatch [1, 2, 3] + [1, 2]
Test Passed
      Thrown: DimensionMismatch
```

- Grouping related tests with the `@testset` macro:

```
using Test

function get_word_lengths(s::String)
    return [length(w) for w in split(s)]
end

@testset "Testing get_word_length()" begin
    text = "Three tomatoes are walking down the street"
    @test get_word_lengths(text) == [5, 8, 3, 7, 4, 3, 6]
    number = 123
    @test_throws MethodError get_word_lengths(number)
end
```

Catch2

- C++
- Project page with installation instructions
- Widely used
- Header-only
- Very rich in functionality
- Well-integrated with CMake

```
#include <catch2/catch.hpp>

#include "example.h"

using namespace Catch::literals;

TEST_CASE("Use the example library to add numbers", "[add]") {
  auto res = add_numbers(1.0, 2.0);
  REQUIRE(res == 3.0_a);
}
```

Google Test

- C++
- Documentation
- Widely used
- Very rich in functionality
- Well-integrated with CMake

```
#include <gtest/gtest.h>

#include "example.h"

TEST(example, add) {
  double res;
  res = add_numbers(1.0, 2.0);
  ASSERT_NEAR(res, 3.0, 1.0e-11);
}
```

- Example project

Boost.Test

- C++
- Documentation
- Very rich in functionality
- Header-only use possible

```
#include <boost/test/unit_test.hpp>

#include "example.h"

BOOST_AUTO_TEST_CASE( add )
{
  auto res = add_numbers(1.0, 2.0);
  BOOST_TEST(res == 3.0);
}
```

---

pFUnit

- Fortran
- Documentation and installation
- Very rich in functionality
- Requires modern Fortran compilers (uses F2003 standard)

```
@test
subroutine test_add_numbers()

   use hello
   use pfunit_mod

   implicit none

   real(8) :: res

   call add_numbers(1.0d0, 2.0d0, res)
   @assertEqual(res, 3.0d0)

end subroutine
```

- Example installation instructions

To test the `factorial` and `fizzbuzz` functions from the test-design exercises, use this `CMakeLists.txt` file:

```
cmake_minimum_required(VERSION 3.12)

project (PFUNIT_DEMO_CR
  VERSION 1.0.0
  LANGUAGES Fortran)

find_package(PFUNIT REQUIRED)
enable_testing()

# system under test
add_library (sut
  factorial.f90
  fizzbuzz.f90
  )

target_include_directories(sut PUBLIC ${CMAKE_CURRENT_BINARY_DIR})

# tests
set (test_srcs test_factorial.pf test_fizzbuzz.pf)
add_pfunit_ctest (my_tests
  TEST_SOURCES ${test_srcs}
  LINK_LIBRARIES sut
  )
```

You can then compile using this script:

```
#!/bin/bash -f

if [[ -d build ]]
then
    rm -rf build
fi

mkdir -p build
cd build
cmake .. -DCMAKE_PREFIX_PATH=$PFUNIT_DIR
make
./my_tests

# or
# ctest --verbose
```

- Example project

Services to deploy testing and coverage

Each of these are web services to handle testing, free for open source projects.

- GitHub Actions (we will demonstrate this in the next episode)
- GitLab CI (we will demonstrate this in the next episode)
- Azure Pipelines
- Coveralls
- Codecov

Good resources

- Getting Started With Testing in Python

Keypoints

- Testing is a basic requirement of any possible language
- There are various tools for any language you may use
- There are free web services for open source

Instructor guide

Detailed schedule

- 9:00-9:15 Motivation
- 9:15-9:40 Testing locally
  - explain the exercise: 5 min
  - **20 min exercise**

- 9:40-10:00 Automated testing
  - demo
- 10:00-10:10 Break
- 10:10-10:50 Test design
  - explain the exercise: 10 min
  - **20 min exercise**
  - discussion and type-along of advanced exercises: 10 minutes
- 10:50-11:00 Discussion and summary

Why we teach this lesson

- Because writing tests and using automated testing is often part of the development process, especially for codes that are larger than a simple script.
- We wish to give learners the tools so that they can rewrite codes and collaborate on a code projects with more confidence.
- Testing can help detecting regressions during development instead of later (when using the code) or too late (after having published results).
- Can simplify collaboration since contributors can be more confident that their changes preserve expected functionality.
- Tests document the intent of the function/module/library/code.
- Tests can guide the code towards more modular and reusable code style with fewer side effects.

Intended learning outcomes

- Know that in each language tools exist that can execute a series of test functions and report success or failure.
- Know that services like GitHub Actions, Azure pipelines, GitLab CI, Appveyor, and others allow us to automatically run tests on every Git push or every pull request or merge request.
- Be able to approach a function or module or library and design a test for it “in words” or pseudo-code.
- Know what test coverage means and how it can help to detect untested/unused code.
- Know where in a larger and untested project to start introducing tests since it is beyond reach and also probably not reasonable to test everything.

How we teach this lesson

How to start

The quote from another testing lesson by Kathryn Huff can be discussed to get participants to reflect on the role of software testing in science and its parallels to calibrating measurement devices. The second half of the quote is left out because it goes so far as to say that untested software does not constitute science, and this can be offending to learners.

Questions to involve participants

- Do you agree that testing software is similar to calibrating detectors?
- Do you test your code?
- How and when do you test?
- Where would you start adding a test in an existing project?
- In which situations would you recommend not to add any tests?

Structure of the lesson

Motivation and concepts

The former “motivation” and “concepts” episodes have been combined into one. Previously, the lesson was described as quite dogmagtic and without enough time for exercises. When teaching, try to give some of the benefits, but without being too long about it. The quick reference has more details on language-specific things.

Testing locally

After introducing the concepts the lesson becomes hands-on in the Testing Locally episode. The pytest example there can be done as type-along and the optional exercise “Testing with numerical tolerance” can be left as homework. The end-to-end test may be hard for many people who can’t script other programs, but it provides something that can keep advanced people interested for longer.

Automated testing (or full-cycle collaborative workflow)

After learning how to run tests locally, the lesson goes into automated testing using GitHub Actions or GitLab CI in the Automated Testing episode. This episode is a simplified version of the collaborative exercise in the optional episode “Full-cycle collaborative workflow”, and the instructor has a choice which one to teach (with the latter requiring around double the time). The exercise in Automated Testing can either be done as a type-along demonstration by the instructor or as an exercise in breakout rooms.

If run as an exercise, this episode requires that learners know how to set up Git repositories locally and online and how to work with pull/merge requests!

If the optional Full-cycle episode is covered, it is useful during the wrap up of the exercise to have a co-instructor doing the exercise in parallel with you. As your partner makes a pull requests with “fixes #1” (or whatever PR number) you can show that the PR links to a issue, and the issue pointing to a commit.

Test design

This episode now has exercises in different languages covering different testing approaches (unit tests, integration tests, testing randomness, TDD). In normal workshops there are far too many exercises for the available time so the instructor has freedom to recommend

specific exercises, to let learners choose themselves, or to recommend learners to discuss rather than implement the tests.

Order of Continuous Integration and Test Design

When this lesson is taught standalone in a software testing workshop or hackathon, it might be better to go through the Test Design episode before the Continuous Integration episode.

Timing

The lesson is stipulated to take around 2 hours, but if more time is available you can replace the Automated Testing episode with “Full-cycle collaborative workflow”, or give more time to the Test Design episode.

Splitting the lesson in two halves with a lunch break in the middle works well. Before lunch you have on hour introduction and the two first exercises. After lunch you continue with the automated testing exercise and test design.

Room for improvement

- The lesson is still too Python centric in the “Testing locally” and “Automated testing” episodes. It would be nice to offer alternatives for learners who write in different languages.
- Currently the lesson is in the “discipline” domain. Testing is something you do to control behavior (your own as well as the code). Testing is also something you can do to improve productivity, but this is less visible/detectable.

Field reports

2024 March (first time with no exercises)

This was the first year we did the “all demo” mode with no exercises, and it worked well enough. Our rough plan can be seen at this hackmd (not pasted here because it’s kind of long).

We tried to be fast with motivation, and not go to details. In “testing locally”, we basically did the exercise as a demo. There wasn’t much of note, it worked and there was plenty of discussion. “Automated testing” worked fine also, except: a) we had to choose how to push to Github, we used vscode, but not all learners did that method. Nothing to change here, but be aware you should explain what and why you are doing for people who didn’t use your chosen method of linking, and b) code coverage didn’t work from a PR from a fork. We had one person set up the automated tests, then swapped instructors and showed the second making the fork and contributing.

The biggest thing to think about is how, when demoing testing locally/automated testing, people ask “how to tests work?”, which then comes later. I don’t know if I would change the order, though… because showing how to design tests without knowing how to use them

would make the same kind of questions, but reversed. I think it's just a "decide how it is and prepare for the questions" - maybe in "motivation" or some other summary episode at the start.

"Test design" didn't have any major problems, but many of the tests were relatively simple to discuss: the earlier tests would be trivial to understand if someone shows you the code (should we demonstrate typing two lines that's the same as in motivation? - but interesting to do as an exercise). Some of the later ones were interesting to build up and demonstrate. It would be good to think which exercises one wants to demo, and which one wants to only talk about. And think which comes out of each demo/telling.

Overall, no major issues in doing this demo-based with good familiarity but limited preparation time.

2023 September

For this lesson, "Motivation" and "Concepts" were combined. An extra end-to-end test was added under "testing locally".

2023 March

Due to a mix-up with instructors, we changed the plan at the last minute and it didn't go so well (we should have known). The original plan had pytest as a demo, with exercises for Github CI and test design. We did pytest as an exercise, Github CI as an exercise, and had almost no time for test design. One complaint was that there wasn't enough time for practical things, so I think a plan like last time could have been good. As usual, we should have found a way to talk less and do more.

This was the schedule from March 2023:

```
* 8:50 - 9:00 Getting started
* 9:00 - 10:45 [Software testing](https://coderefinery.github.io/testing/)
  - 9:00-9:05 Short info about today's exercise sessions and possible questions from
yesterday
  - 9:05-9:10 [Motivation](https://coderefinery.github.io/testing/motivation/)
  - 9:10-9:20 [Concepts](https://coderefinery.github.io/testing/concepts/)
  - 9:20-9:45 [Testing locally](https://coderefinery.github.io/testing/pytest/)
      - 5 minutes talking
      - 15 minute exercises
      - 5 going over exercises and discussion
  - 9:45-9:55 Break
  - 9:55-10:20 [Automated testing](https://coderefinery.github.io/testing/continuous-
integration/)
      - type-along session
  - 10:20-10:45 [Test design](https://coderefinery.github.io/testing/test-design/)
      - discussion: 5 minutes
      - Exercises: 10 minutes.  Any of exercises Design-1 to Design-8 that learners
want to do.
      - discussion and type-along of advanced exercises: 10 minutes
  - 10:45-11:00 Break
```

2022 September

Overall, it went well with the schedule below. We had to be efficient, but it seemed like it wasn't too hard. The only complaint was that there was a lot of time for the first exercise (pytest locally), which I would agree with. I think for next time, we should add in parts of that exercise so that there are more advanced things to do, since there are more nice features of pytest, such as `--pdb`, running single tests, etc. Other than that, there wasn't really much to report.

The way extra long breaks and a 5-minute starting point were included is quite good for the schedule. Not because we needed them, but it allowed us to go over time and not be under so much time pressure.

```
**Day 6**
* 8:50 - 9:00 Getting started
* 9:00 - 10:45 Software testing
  - 9:00-9:05 Short info about today's exercise sessions and possible questions from
yesterday
  - 9:05-9:10 [Motivation](https://coderefinery.github.io/testing/motivation/)
  - 9:10-9:20 [Concepts](https://coderefinery.github.io/testing/concepts/)
  - 9:20-9:45 [Testing locally](https://coderefinery.github.io/testing/pytest/)
      - 5 minutes talking
      - 15 minute exercises
      - 5 going over exercises and discussion
  - 9:45-9:55 Break
  - 9:55-10:20 [Automated testing](https://coderefinery.github.io/testing/continuous-
integration/)
      - type-along session
  - 10:20-10:45 [Test design](https://coderefinery.github.io/testing/test-design/)
      - discussion: 5 minutes
      - Exercises: 10 minutes.  Any of exercises Design-1 to Design-8 that learners
want to do.
      - discussion and type-along of advanced exercises: 10 minutes
  - 10:45-11:00 Break
```

2019 November instructor training workshop

Some notes about teaching are here: https://github.com/coderefinery/testing/issues/42