SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 14

Fabian Wermelinger

Harvard University

CS107 / AC207

Tuesday, October 18th 2022

LAST TIME

- Dual numbers exercise
- Implementation approaches for automatic differentiation
- Operator overloading
- Reverse mode AD
- Examples for application and project extensions

TODAY

Main topics: ***Continuous integration (CI), testing code***

Details:

- Continuous Integration (CI) in Software Development
- GitHub actions
- Testing code
- Test-Driven development

AGENDA CHECK:

- Milestone 1 is due on Thursday. This milestone is about *explicit design* (recall explicit/implicit software design approaches lecture 7/8). Try to define a basis you want to build your library on. You can still change design choices later, but keep in mind that this becomes harder and harder as your code starts to become larger and larger.

CONTINUOUS INTEGRATION (CI)

Continuous Integration (CI) is a *software development* process where developers integrate new code (for example Git commits) into an ***automated testing and documentation pipeline*** that streamlines the build and deploy procedure of a project and helps to ***detect*** errors and bugs early in the introduction phase.

- CI significantly improves quality in software development.
- A version control system (VCS) is at the heart of a CI pipeline.
- Automating tests and generation of documentation are ***essential*** in any serious code base.
- Understanding how CI works requires combined knowledge of how a shell works, VCS and containerization (e.g. podman or docker).

CONTINUOUS INTEGRATION (CI)

How does a CI workflow look like?

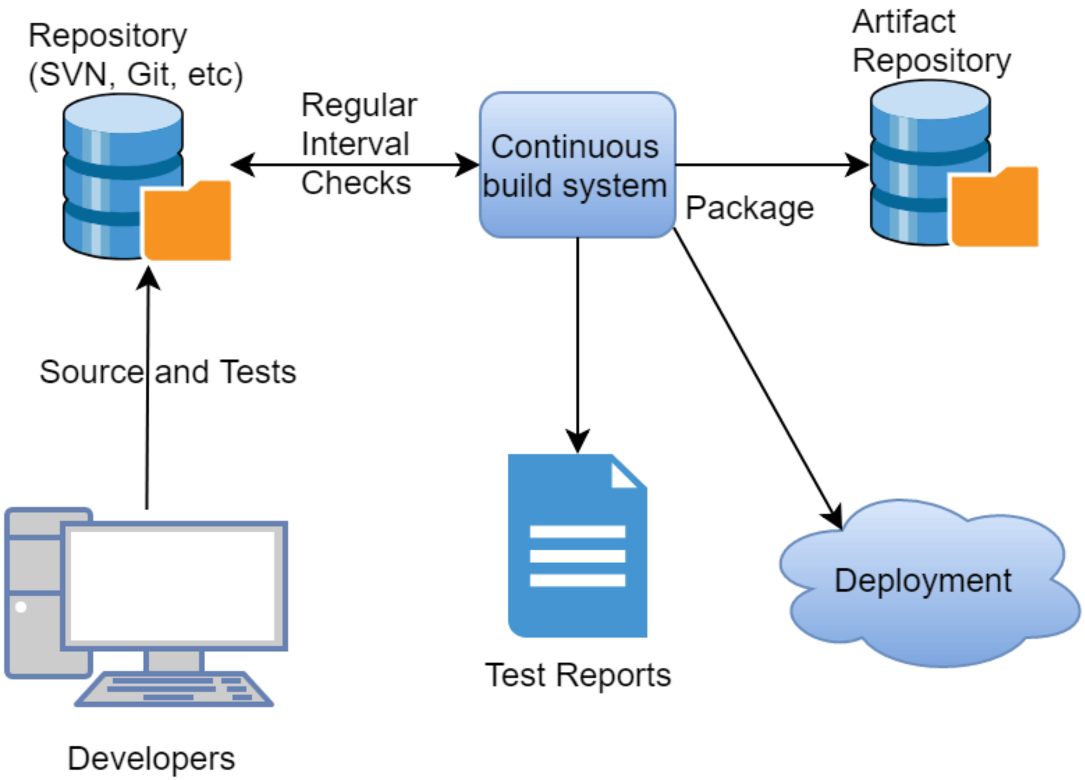

Image taken from https://www.brightdevelopers.com/what-is-jenkins-and-why-it-is-so-important

- Source code and code for testing belongs in your VCS.
- A CI system frequently checks a remote repository for new commits. Alternatively, a service like GitHub can *trigger* a CI system as new commits are being pushed.
- The CI system generates reports of several tasks and informs the developers about status through channels like email, messenger integration (e.g. slack) or other means.
- The various tasks may generate output that is not necessary for successful completion of the CI pipeline but can be useful for debugging. This data is called an "*artifact*" and would need to be stored somewhere (requires resources). *Such a service is optional.*

CONTINUOUS INTEGRATION (CI)

What is inside a CI system?

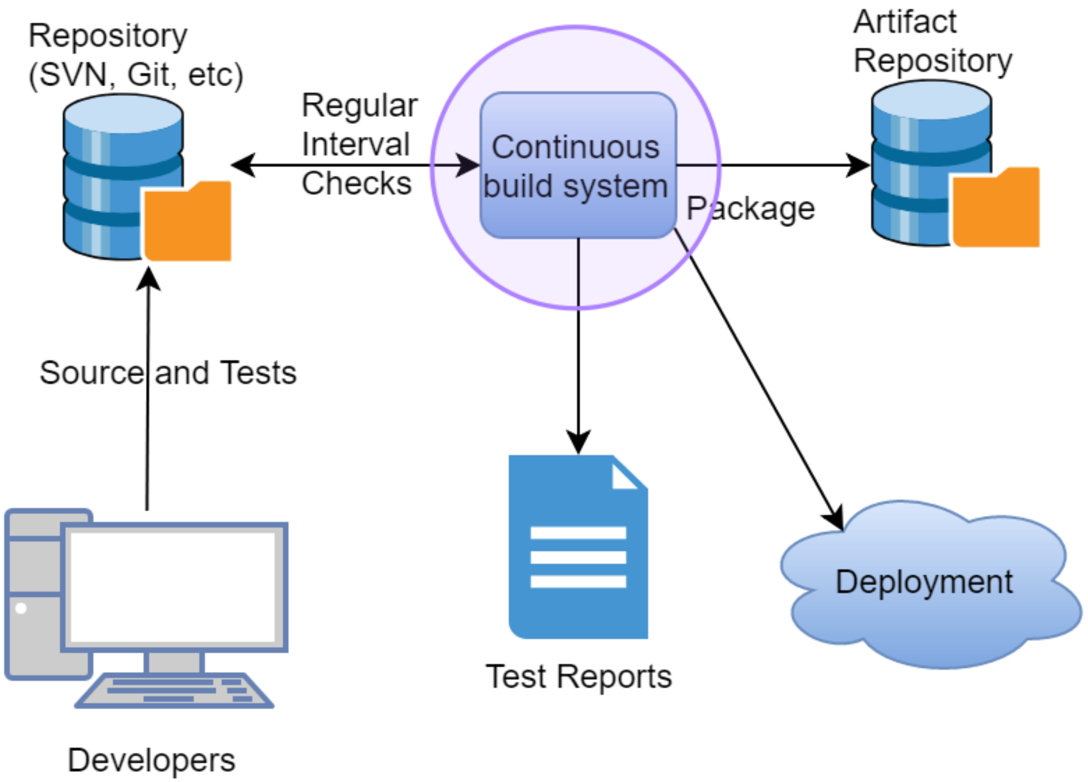

Image taken from https://www.brightdevelopers.com/what-is-jenkins-and-why-it-is-so-important

- In essence a CI system is a server that will launch a ***build*** of your project according to some rules that you have configured.
- Because these rules can be extensive, a CI server must offer ***flexibility with respect to the build platform*** → ***this flexibility is achieved through containerization!***
- What are the main tasks to be performed in CI? Depends on your needs but most often you require ***testing***, ***documentation*** and ***deployment*** to be automated. *Additionally, testing may require:*
  - Quality assessment of tests (***coverage***)
  - Building code with an assortment of compilers on various systems like Linux, MacOSX, Windows (including different versions of them)
  - Running benchmarks and profiling reports

CONTINUOUS INTEGRATION (CI)

Requirements on CI:

- ***Reports about build/test success or failure should be received instantly:***
  - You should always run ***unit tests***. These are cheap small units that test the core functionality and interfaces of your code. If possible, ***integration tests*** should be executed for each build as well.
  - More expensive test suites (in terms of time and resources) like integration tests and acceptance tests may be scheduled over night instead for every commit.
- Your CI configurations may define many tasks to be run for every triggered event. This will require to execute these tasks *in parallel* (default for GitHub actions → *you must define dependencies if any*).
- Output generated from your tasks may need to be stored for post-processing (debugging, trouble-shooting or archiving).

All of this requires considerable computational resources. You will either need to acquire hardware or invest in a hosted service to run CI.

CONTINUOUS INTEGRATION (CI)

Some commonly used CI platforms:

There are many CI providers, this is a non-preferential selection of few:

- https://www.appveyor.com
- https://azure.microsoft.com/en-us/services/devops/pipelines
- https://bitbucket.org/product/features/pipelines
- https://circleci.com
- https://github.com/features/actions *(what we use in class)*
- https://about.gitlab.com/stages-devops-lifecycle/continuous-integration
- https://www.jenkins.io
- https://www.travis-ci.com

→ Jenkins and GitLab are software solutions that you can use to run your own CI server. GitLab offers limited *free* features.

CONTINUOUS INTEGRATION (CI)

What are the basic steps performed in a CI run?

(The following may deviate slightly depending on the CI provider but they all achieve the same goal)

Most importantly: your CI builds run inside a virtual environment, e.g. a Docker container. Some configuration/customization may be needed prior to running the first build.

1. The CI process clones your VCS repository into the Docker container and switches to the corresponding commit to be tested.
2. Compile and/or install your software project. This process should be supported by a build automation system of your choice. Examples are `make`, `cmake`, `meson` or `setuptools` (for Python anything that supports PEP517 will work nicely).
3. Execute a chain of (possibly dependent) jobs that will run tests using the built/installed software. Independent jobs may run in parallel while others depend on completion of preceding jobs. ***A build is considered successful if all jobs in the chain exit with a success exit code `0`.***
4. Post-processing depending on success or failure of the job chain. This could include deploying releases to PyPI or generating test coverage reports, for example.

CI EXAMPLE: GITHUB ACTIONS

- CI requires a configuration somewhere in your project root. The exact place where this configuration should be varies for different CI systems.
- The language to specify this configuration should be well structured and well readable. Typically YAML (https://yaml.org/) is used.
- All the CI setup, jobs and dependencies are defined in such YAML files.
- You have already met GitHub actions in project milestone 1B:
  - You have configured *workflows*. A workflow (GitHub action specific) is an automated process that runs one or more jobs.
  - Workflows are configured in YAML files and stored in a `.github/workflows` directory. (The `.github` directory is in your project root.)
  - Workflows are triggered when some *event* occurs. Such trigger(s) are configured in the YAML file for the workflow.

CI EXAMPLE: GITHUB ACTIONS

Terminology:

Workflows

A workflow is a configurable automated process that will run one or more jobs.

Events

An event is a specific activity in a repository that triggers a workflow run.

Jobs

A job is a set of steps in a workflow that execute on the same runner.

Actions

An action is a custom application for the GitHub Actions platform that performs a complex but frequently repeated task.

Runners

A runner is a server that runs your workflows when they are triggered.

Example workflow:

```yaml
# Name of the workflow (optional)
name: Lecture14 Continuous Integration

# Controls when the workflow will run
on:
  # Triggers the workflow on push or
  # pull request events only for the
  # `main` branch
  push:
    branches:
      - main
  pull_request:
    branches:
      - main
  # Allows to run this workflow
  # manually from the Actions tab
  workflow_dispatch:
```

The `on:` key defines when the workflow is triggered.

CI EXAMPLE: GITHUB ACTIONS

Example workflow:

```yaml
# Name of the workflow (optional)
name: Lecture14 Continuous Integration

# Controls when the workflow will run
on:
  # Triggers the workflow on push or
  # pull request events only for the
  # `main` branch
  push:
    branches:
      - main
  pull_request:
    branches:
      - main
  # Allows to run this workflow
  # manually from the Actions tab
  workflow_dispatch:
```

The `run:` key defines the command(s) to be run for a step.

Jobs:

- This workflow does not run anything.
- *Jobs* must be defined for this purpose.
- A workflow can have multiple jobs, where each job consists of *step*s.
- Because the jobs run in a Docker container (Ubuntu here), you can use any shell commands that exist in the container.

CI EXAMPLE: GITHUB ACTIONS

Example workflow:

```
# Name of the workflow (optional)
name: Lecture14 Continuous Integration

# Controls when the workflow will run
on:
  # Triggers the workflow on push or
  # pull request events only for the
  # `main` branch
  push:
    branches:
      - main
  pull_request:
    branches:
      - main
  # Allows to run this workflow
  # manually from the Actions tab
  workflow_dispatch:
```

Actions:
https://github.com/marketplace?type=actions

Jobs:

- What if the command in the `run:` key was a shell script in your repository?
- The job may need to checkout a specific branch → some additional step is required.
- GitHub ***actions*** are tasks that are frequently required. Checking out a repository is one of them.
- Actions are like "plugins" → `uses:` them in your workflows. ← There are *many* available!

CI EXAMPLE: GITHUB ACTIONS

- The workflow *fails* when a job returns a non-zero exit code. (A zero exit code typically indicates success.)
- An example would be a test in your test suite or tests that depend on each other, etc.
- By default, the GitHub (enterprise) platform will send an email to notify that workflows have failed.
- The runners used when you push code to https://code.harvard.edu are *self-hosted* by Harvard University and you should be able to run CI as often you like.
- In practice you would have to buy this service from some provider or buy your own hardware and setup a self-hosted system.

TESTING

- One workflow you must install in ***every*** software project is to run an extensive test suite.
- There are different ways to achieve this in Python.
- The most accepted solutions are `pytest` (external Python package that must be installed) or `unittest` which is buit-in to the Python distribution.
- *An example where code is tested this way are the homework autograders:*
    - When a student uploads a submission, the CI triggers and runs the autograder job in a Docker container (one for every submission).
    - The test code (students work) is imported and a number of tests are performed.

TESTING

- Testing your code gives you confidence that the expected behavior is observed ***without side-effects***.
- Nobody will consider using your code if there are no tests associated with it. ***You must be rigorous here!*** *Show your peer that you mean business!*
- ***What should you test then?***
  - ***Recall:*** OOP is about data encapsulation, inheritance and polymorphism. These are the internals (implementation) which are accessible through an ***interface***.
  - Typically the interface requirements are specified in a Software Requirements Specification (SRS) → the SRS is an ***explicit*** approach, it establishes a *contract* between you and your customer(s).
  - Your test suites must ensure that the software requirement specifications are met according to the contract.

TESTING

Example for an interface:

- Assume you are working on a library for complex numbers.

```
>>> from your_library import Complex
>>> z1 = Complex(1, 1)
>>> z2 = Complex(2, 2)
>>> z3 = z1 * z2
```

- There are three interfaces in the code above:
    1. The `import` statement
    2. Instance creation of `Complex` type (`__init__`)
    3. The multiplication operator (`__mul__`)
- The `import` statement will be tested implicitly when you use it in your test suites (otherwise importing `Complex` would fail). The `__init__` and `__mul__` interfaces must be tested explicitly by writing tests.

TESTING

How to write tests: two possibilities

1. ***First write the tests*** (according to the requirements in the SRS) and then the implementation of your interfaces (***black box*** tests).
2. ***First write the implementation of your interfaces*** (according to the requirements in the SRS) and then the tests (***white box*** tests).

Are there problems associated with either of the two?
How are duck-typing and white box tests related?

→ ***Test-Driven Development*** (TDD) is a manifestation of black box testing. It is a software design strategy that relies on first writing the SRS (explicit design). *Tests are then written following the SRS before you start with the implementation*. This approach is ***hard*** and requires considerable amount of time to carefully write the tests!

TESTING

Test-Driven Development (TDD):

- Is a *predictable* way to develop. You know when you are finished (when the test passed), without having to worry about bugs associated to the code you tested.
- It gives you a chance to *learn all of the lessons that the code has to teach you*. If you only slap together the first thing you think of, then you never have time to think of a second, better thing (e.g. autograded homework).
- It *improves* the lives of the users of your software.
- It lets your team mates count on you, and you on them.
- It feels good to write the passing code.

TESTING

There are different types of tests:

- ***Unit tests:*** these are the smallest tests applied to classes and functions in a module and sometimes a module itself. *Can be black box tests, often realized as white box tests.*
- ***Integration tests:*** these tests combine different units that have a dependency on each other. Unit tests alone can not guarantee a correct interdependency among units.
- ***Regression tests:*** after integration testing (and possibly fixing errors) regression tests are conducted which re-run the unit tests to ensure that integration did not break any of the core functionalities.
- ***System and Acceptance tests:*** these are usually larger tests that take place upon multi-module completion which compose a part or the whole of a software system. Acceptance tests involve the ***customer*** who provides feedback on the test results. Acceptance tests should be carried out early on to account for customer feedback iteratively. *(Customers are demanding!) System and acceptance tests should be black box tests based on the SRS.*

TESTING

What should be tested?

Test simple (and often trivial) parts with unit tests.

- Add integration tests when there are dependencies among units.
- Your system and acceptance tests will fail at the beginning (if they would not it means your work is complete).
- Make sure your unit and integration tests are executed in your CI builds.
- Whenever you fix integration tests, re-run your tests locally to enforce regression. Frequent committing will also trigger regression through the continuous integration (CI).
- *Code defensively*: add test code that handles the "can't happen" case. This is what is meant by "trivial" above. Even if you think it is nonsensical to test a trivial statement, Murphy's law will prove you wrong! ***Examples:*** zero-length arrays for inputs or integer overflow.
- *Test code at its boundaries*: this is where most failures happen. ***Examples*** include empty inputs, too many inputs, wrong input types or order of expected input is wrong.

TESTING IN PYTHON

- Python provides a few packages in the standard library (https://docs.python.org/3/library/development.html) that are useful for testing.
- `unittest`: unit testing framework
- `doctest`: a test module that utilizes *docstrings* for testing. (Docstrings are covered when we talk about documentation.)
- `pytest`: a useful testing framework outside the Python standard library. It is compatible to run tests written with the `unittest` package.

PYTHON `unittest`

The `unittest` framework is a simple Python package that uses a set of assert methods to test code. Tests *inherit* a unit test base class.

(For C++ → good testing frameworks are `googletest`, `catch2` or `doctest` if the project is small)

Anatomy of unit tests:

- ***Recall:*** a unit test *is small* and addresses functions, classes and interfaces. It is a good idea to write unit tests for individual modules (its contents) in your project. *Unit test suites should complete in matter of seconds.*
- How you organize your tests is up to you. There is no definite rule.

  (***My suggestion:*** tests should be *separate* from source code → *Some disagree.*)
- ***Example:*** recall our test project:

```
cs107_project
├── src
└── tests              # all tests are contained in here
    ├── run_tests.sh   # optional: convenience test harness to run tests
    └── subpkg_1
        └── test_module_1.py # tests are modules (this one tests subpkg_1.module_1.py)
```

PYTHON unittest

- ***Example:*** recall our test project:

```
cs107_project
├── src
└── tests              # all tests are contained in here
    ├── run_tests.sh   # optional: convenience test harness to run tests
    └── subpkg_1
        └── test_module_1.py # tests are modules (this one tests subpkg_1.module_1.py)
```

- ***Convention:*** name your test (modules) the same as the modules you test and prepend the filename with "`test_`". This will allow tools to automatically find your tests.
- Here test organization mirrors the directory structure used in source code → *entirely up to you.*
- You may use a *driver* script (test harness) that conveniently runs your tests (handy for CI later on) → *shell scripts are a powerful tool for this!*
- ***You must write tests as serious as you write general code. They are not optional but an integral part of any software project!***

PYTHON unittest

Anatomy of a Python `unittest`:

```python
"""
This test suite (a module) runs tests for subpkg_1.module_1 of the cs107_package.
"""
import unittest  # Here we use `unittest` from the Python standard library

# project code to test (requires `cs107_package` to be in PYTHONPATH)
from cs107_package.subpkg_1.module_1 import (Foo, foo)

class TestTypes(unittest.TestCase):  # test classes must inherit from unittest.TestCase
    # test methods (functions) must be prepended with `test_`.
    def test_class_Foo(self):
        """
        This is just a trivial test to check that `Foo` is initialized correctly.
        More tests associated to `Foo.__init__` could be written here.
        """
        f = Foo(1, 2)  # create instance
        self.assertEqual(f.a, 1)  # check attribute `a`
```

- Use *classes to create a high-level structure* for your tests. Use member functions to test individual features.
- You test your code by calling different `self.assert*` methods inherited from `unittest.TestCase`.
- Python conventions on test discovery: https://docs.pytest.org/en/6.2.x/goodpractices.html#conventions-for-python-test-discovery

PYTHON `unittest`

How to run Python `unittest`'s?

- If you are executing the `unittest.main()` function in your executable test module code, then you can just execute the module.
- It is not necessary to add this code in your test modules. For example, you can also run a *specific* test with:

```
$ python -m unittest subpkg_1/test_module_1.py
```

- If you want to *auto-discover* tests you can run without a specific test module:

```
$ python -m unittest
```

(must be executed *in the directory* where the test modules are)

- ***Auto-discovery only works if you follow the naming convention!***

PYTHON `unittest`

How to run Python `unittest`'s?

- It is generally up to you how you organize to run your tests. *The simplest strategy is to rely on auto-discovery*.
- ***You want flexibility*** → create a test driver shell script:
  1. It should be easy to add new tests or quickly comment tests out. Aim at a ***modular*** design.
  2. You may want to be generic with your driver script such that you can ***wrap*** multiple Python test tools around it.
  3. The driver script must run on both, your local development platform and also in a CI container.
- A shell script can achieve all this easily. Be careful with `zsh` or other shells here because some CI containers may not support it (if you use specific shell commands). Use `sh` or `bash` compatible scripts (those have stood the test of time).

PYTHON unittest

Example test harness script: (see `run_tests.sh`)

```bash
#!/usr/bin/env bash

# list of test cases you want to run
tests=(
    # test_other_things_on_root_level.py
    subpkg_1/test_module_1.py
    subpkg_1/test_module_2.py
    # TODO: subpkg_2/test_module_3.py
)

# Must add the module source path because we use `import cs107_package` in
# our test suite.  This is necessary if you want to test in your local
# development environment without properly installing the package.
export PYTHONPATH="$(pwd -P)/../src":${PYTHONPATH}

# decide what driver to use (depending on arguments given)
```

PYTHON pytest

- The `unittest` package works well as a general testing framework and it is available in the Python standard library.
- It is limited to writing tests in classes using various `self.assert*` methods (https://docs.python.org/3/library/unittest.html#assert-methods) which can be difficult to remember.
- The `pytest` package is an alternative to `unittest` and typically the *preferred choice* of testing framework:
  - Instead of `self.assert*`, `pytest` only requires the default Python `assert` statement for all tests.
  - It is *compatible* with tests written using the `unittest` package.
  - You can test *standalone functions* or group tests into `TestClasses` like we do for `unittest`.
  - For simple tests, importing `pytest` is not necessary.

PYTHON pytest

Anatomy of a Python pytest: (`python -m pip install pytest`)

```python
"""
This test suite (a module) runs tests for subpkg_1.module_2 of the cs107_package.
"""

# project code to test (requires `cs107_package` to be in PYTHONPATH)
from cs107_package.subpkg_1.module_2 import (bar)

class TestFunctions:
    """We do not inherit from unittest.TestCase for pytest's!"""

    def test_bar(self):
        """
        This is just a trivial test to check the return value of function `bar`.
        """
        # assert the return value of bar() (note that this uses Python's
        # `assert` statement directly, no need to inherit from anything!)
        assert bar() == "cs107_package.subpkg_1.module_2.bar()"
```

- Use *classes to create a high-level structure* for your tests. Use member functions to test individual features.
- You test your code using the built-in `assert` statement exclusively.
- Python conventions on test discovery apply for `pytest` as well.

PYTHON pytest

How to run Python `pytest`'s?

- As before with `unittest`, but use the `pytest` executable this time:

```
$ python -m unittest subpkg_1/test_module_1.py  # unittest module from earlier
$ pytest subpkg_1/test_module_1.py              # pytest
```

- ***Note:*** `pytest` will recursively search for tests. To run all tests:

```
$ pytest
```

> This will run all discoverable tests for both, tests written with the `unittest` module ***or*** `pytest`! `pytest` is *compatible* with `unittest`.

- Alternatively, the test harness script can use `pytest` as well:

```
$ ./run_tests.sh pytest -v
```

> The benefit of using a test harness script is that the environment is setup correctly (i.e. `PYTHONPATH` environment variable).

CI EXAMPLE: GITHUB ACTIONS

Create a workflow for testing:

```yaml
# Name of the workflow (optional)
name: Lecture14 Continuous Integration Tests

# Controls when the workflow will run
on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main
  workflow_dispatch:

jobs:
  install_and_test: # job ID
    runs-on: ubuntu-latest # The type of runner that the job will run on

```

- The ingredients in this workflow are all things we have discussed up to here: shell scripts, building a Python package using PEP517 and PEP518, running tests with `pytest`.
- All of it happens fully automated when the workflow is triggered → ***Test-Driven development!***

RECAP

- Continuous Integration (CI) in Software Development
- GitHub actions
- Testing code
- Test-Driven development

Further reading:

- Beck, K., *Test-Driven Development: By Example*, Addison-Wesley 2003
- Continuous integration (CI): https://en.wikipedia.org/wiki/Continuous_integration
- Containerization: https://www.ibm.com/cloud/learn/containerization
- `pytest`: https://docs.pytest.org/en/7.1.x/
- `unittest`: https://docs.python.org/3/library/unittest.html