CS107 / AC207

SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 14

Thursday, October 21st 2021

Fabian Wermelinger
Harvard University

RECAP OF LAST TIME

- Virtual machines
- Virtual environments
- Docker containers

OUTLINE

- Continuous Integration (CI) in Software Development
- Testing your code and verifying the *quality* of your tests
- Documentation

CONTINUOUS INTEGRATION (CI)

CONTINUOUS INTEGRATION (CI)

Continuous Integration (CI) is a *software development* process where developers integrate new code (i.e. 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?

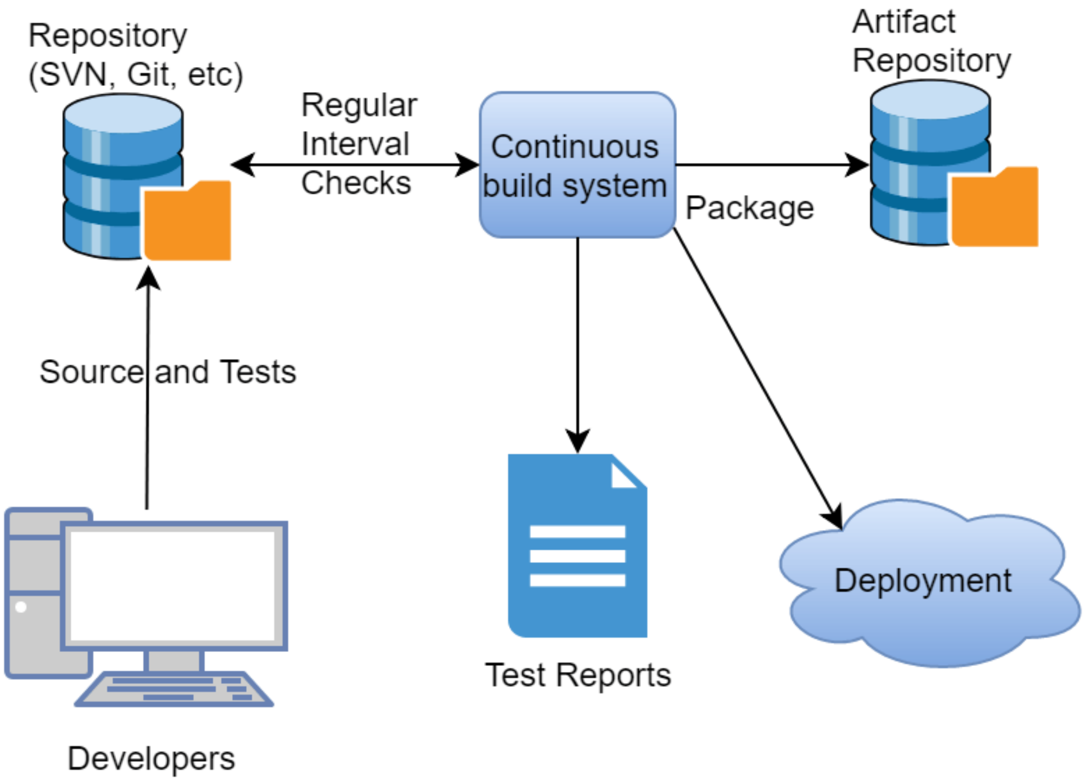


Image taken from https://www.brightdevelopers.com/what-is-jenkins-and-why-it-is-so-important

- 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?

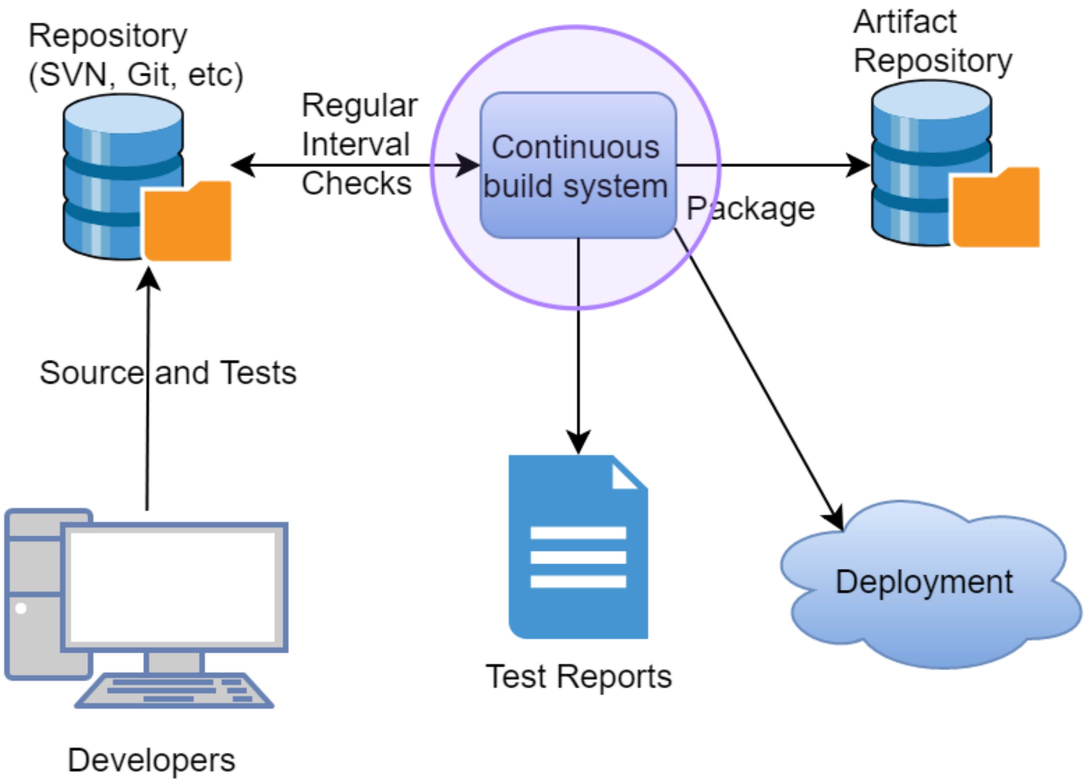


Image taken from https://www.brightdevelopers.com/what-is-jenkins-and-why-it-is-so-important

- 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 these "rules"? Defined by your needs but *testing*, *documentation* and *deployment* are important rules you will need. Extensions to testing may include:
  - 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:

- You want to receive build reports almost 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 (time and resources) like (possibly) integration tests and acceptance tests may be scheduled over night or higher frequency.
- As you may have many CI rules defined, you possibly need to execute your rules *in parallel*.
- Generated output from your rules may need to be stored for inspection (debugging or trouble-shooting).

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

CONTINUOUS INTEGRATION (CI)

What are 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
- 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 where the latter offers limited free features.

CONTINUOUS INTEGRATION (CI)

What steps are performed in a CI build?

The following may deviate slightly depending on the CI provider. From the bird's eye perspective they implement the same.

Your CI builds run inside a virtual environment (e.g. a `docker` container). Some configuration is usually needed to set them up before running the build.

A build consists of the following:

1. The CI process clones your VCS repository into the 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` or anything else that supports PEP517/518 for `python` specific projects.
3. Define a dependency chain of jobs that will run the built or 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 success.***
4. Post-processing depending on success or failure of the job chain. (This could include deploying releases to PyPI for example.)

CI EXAMPLE: TRAVIS-CI

- Many CI providers require a configuration file in your project root. In there you define the rules and job chains you want to execute.
- For our example `cs107_package` (Lecture 08), a configuration could look like this (inside a `.travis.yml` file for TravisCI):

```yaml
# Configuration file for TravisCI builds
language: python
jobs:
  include:
    - name: "Python 3.8.0"
      python: 3.8
    - name: "Python 3.9.0"
      python: 3.9

# These packages we only want in our CI builds (for building our package)
before_install: python -m pip install build

# This installs our project in the running container, convenient with PEP517 and
# setuptools (Lecture 08).  We can do this local as well, without uploading to PyPI first
install: (python -m build --wheel && python -m pip install dist/*)

# Run the tests defined in the package
script: (cd tests && ./run_tests.sh) # run tests
```

- This runs two jobs in parallel, one for `python3.6` and another for `python3.8`.
- The installation of our package is trivial since we use `setuptools` via PEP517/518.

CI EXAMPLE: TRAVIS-CI

- For our example `cs107_package` (Lecture 08), a configuration could look like this (inside a `.travis.yml` file for TravisCI):

```
# These packages we only want in our CI builds (for building our package)
before_install: python -m pip install build

# This installs our project in the running container, convenient with PEP517 and
# setuptools (Lecture 08).  We can do this local as well, without uploading to PyPI first
install: (python -m build --wheel && python -m pip install dist/*)

# Run the tests defined in the package
script: (cd tests && ./run_tests.sh) # run tests
```

- The installation of our package is trivial since we use `setuptools` via PEP517/518 (see Lecture 8). We have a build dependency on the `build` package which we resolve with the `before_install` rule.
- If the package has ***runtime dependencies***, declare them in the `setup.cfg` file:

```
[options]
package_dir =
    = src
packages = find:
python_requires = >=3.6
install_requires =
    numpy # if the package depends on numpy declare it here
```

  You do not need a fragmented `requirements.txt` file when using PEP517.
- The last `script:` line executes a ***test suite*** in all jobs we defined earlier.

TESTING

Testing in the CI containers happens when this line is executed in the `.travis.yml` file.

```
# Run the tests defined in the package
script: (cd tests && ./run_tests.sh) # run tests
```

This is how the directory structure of our `cs107_package` looks today (compare with Lecture 8: we have added incomplete tests here):

```
cs107_project/
├── LICENSE
├── pyproject.toml
├── README.md
├── setup.cfg
├── src
│   └── cs107_package
│       ├── __init__.py
│       ├── __main__.py
│       ├── subpkg_1
│       │   ├── __init__.py
│       │   ├── module_1.py
│       │   └── module_2.py
│       └── subpkg_2
│           ├── __init__.py
│           ├── module_3.py
│           ├── module_4.py
│           └── module_5.py
└── tests
    ├── run_coverage.sh
    ├── run_tests.sh
    └── subpkg_1
        └── test_module_1.py
```

TESTING

- Testing your code gives you confidence that the expected behavior is observed without side-effects.
- Nobody (sane) will look or even use your code if there are no tests associated with your work. *Science is rigorous*, 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 *interfaces*.
- Typically the requirements on interfaces are specified in a Software Requirements Specification (SRS). (Recall the discussion about *explicit* software design and implicit duck-typing. The SRS is *explicit*, it establishes a *contract* with your customer(s).)
- Your test suites must ensure that the software requirement specifications are met according to the contract.

TESTING

Example for interfaces:

- 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. The `__init__` and `__mul__` interfaces must be tested explicitly.

TESTING

How to write tests?

You can write your tests in two ways:

1. Write the tests first (according to the requirements in the SRS) and then the implementation of your interfaces (***black box*** tests).
2. 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 a SRS being developed first (explicit design) and tests are written following the SRS before you start with the implementation.

TESTING

There are different levels of testing:

- ***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 based on the SRS.

TESTING

What to test?

- 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 CI.
- Program defensively: add test code that handles the "can't happen" case. This is what is meant by "trivial" in the first item. Even if you think it is nonsensical to test a trivial statement, Murphy's law will prove you wrong! Examples are zero-length arrays or integer overflow.
- Test code at its boundaries: this is where most errors happen. Examples include empty inputs, too many inputs or wrong input types.

TESTING IN `python`

`python` provides a few packages in the standard library (development tools section) that are useful for testing:

- `unittest`: unit testing framework
- `doctest`: a test module that utilizes *doc-strings* for testing. (Doc-strings are covered in the following section.)
- `pytest`: a useful testing framework outside the `python` standard library. It is compatible to run tests written with the `unittest` package.

unittest

The `unittest` framework is a simple `python` package that uses a set of assert methods that you use for testing your code.

(For `C++` good testing frameworks are `googletest`, `catch2` or `doctest` (for small projects)).

Anatomy of a `python unittest`:

- ***Recall:*** a unit test is small and addresses functions, classes and interfaces. It is a good idea to write these tests for individual modules in your code.
- How you organize your tests is up to your liking. You should have them *separate* from your source code.
- For our example toy project:

```
cs107_project/
├── src
└── tests
    ├── run_tests.sh
    └── subpkg_1
        └── test_module_1.py
```

unittest

- For our example toy project:

```
cs107_project/
├── src
└── tests
    ├── run_tests.sh
    └── subpkg_1
        └── test_module_1.py
```

- ***Convention:*** name your tests as your modules and prepend the file name with "`test_`".
- I have chosen to organize the tests using the same directory structure as in the source code. *How you organize your tests is entirely up to you. Be reasonable.*
- Use a simple *driver* script that runs all your tests. Ideally you want it adaptive such that you can exploit multiple testing facilities offered by `python` with one driver script only.
- When you deploy a production release to your customer or to PyPI, test cases and other *development* related data ***are not*** shipped with the release. When you order a carrot salad in a restaurant, the chef will not serve you the peel. (If your project is open-source, this data is always accessible through your public `git` repository.)

`unittest`

How to run tests?

- Entirely up to you! You have some powerful tools in your backpack now to realize a test driver.
- You want flexibility:
    1. It should be easy to add new tests or quickly comment tests out. Keyword here is ***modularity***.
    2. You may want to be generic with your driver script, such that you can ***wrap*** multiple tools around it.
    3. Your driver script must run on your local development platform, but also in a CI container.
- A *shell script* can work perfectly for this task. But be careful with `zsh` or other shells here because some CI containers may not like it. Use `sh` or `bash` compatible scripts (those have stood the test of time).

unittest

Example: how to run tests?

Contents of `./tests/run_tests.sh` (***recall:*** we have configured our `.travis.yml` CI builds to execute this driver):

```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_2/test_module_3.py
)

# decide what driver to use (depending on arguments given)
unit='-m unittest'
if [[ $# -gt 0 && ${1} == 'coverage' ]]; then
    driver="${@} ${unit}"
elif [[ $# -gt 0 && ${1} == 'pytest'* ]]; then
    driver="${@}"
else
    driver="python ${@} ${unit}"
fi

# we must add the module source path because we use `import cs107_package` in our test suite and we
# want to test from the source directly (not a package that we have (possibly) installed earlier)
export PYTHONPATH="$(pwd -P)/../src":${PYTHONPATH}

# run the tests
${driver} ${tests[@]}
```

`unittest`

Example: how to run tests?

- When we run this script without arguments it will execute

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

  The same syntax as you would execute other `python` packages. This runs test cases for `module_1.py` (***recall:*** convention for naming test cases start with `test_`)

- Running the test driver gives:

  ```
  $ ./run_tests.sh
  ..
  ----------------------------------------------------------------------
  Ran 2 tests in 0.000s

  OK
  ```

  Two tests have been run, let's look at them.

unittest

./tests/subpkg_1/test_module_1.py

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

# project code (import into this namespace)
from cs107_package.subpkg_1.module_1 import *

class TestTypes(unittest.TestCase):
    def test_class_Foo(self):
        """
        This is just a trivial test to check that `Foo` is initialized
        correctly.  More tests associated to the class `Foo` could be written in
        this method.
        """
        f = Foo(1, 2)
        self.assertEqual(f.a, 1)
        self.assertEqual(f.b, 2)

class TestFunctions(unittest.TestCase):
    def test_function_foo(self):
        """
        This is just a trivial test to check the return value of function `foo`.
        """
        self.assertEqual(foo(), "cs107_package.subpkg_1.module_1.foo()")

if __name__ == '__main__':
    unittest.main()
```

(The last two lines allow you to execute your test module as a standalone program.)

See this link for the `python` conventions on test discovery:
https://docs.pytest.org/en/6.2.x/goodpractices.html#conventions-for-python-test-discovery

- In `unittest`'s you create `class`es that inherit from `unittest.TestCase`.
- You can use these classes to organize your tests.
- Each class defines *methods* for the tests. They must again start with `test_`. The class type must start with `Test`.
- You test your code by calling different `self.assert*` methods (inherited).
- The two tests ran before correspond to the `test_*` methods in each of the two classes.

pytest

- The `unittest` package works well as a general testing framework.
- You are somewhat limited to writing your tests in classes and you have to remember the various `self.assert*` methods. See https://docs.python.org/3/library/unittest.html#assert-methods for a list.
- The `pytest` package can be an alternative for testing:
  - Instead of `self.assert*`, `pytest` just uses 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`.
- ***Install:*** `python -m pip install pytest`

pytest

We create a new test module: ./tests/subpkg_1/test_module_2.py

```python
"""
This test suite (a module) runs tests for subpkg_1.module_2 of the cs107_package.
"""
import pytest # these tests are designed for pytest

# project code
from cs107_package.subpkg_1.module_2 import *

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 bar() == "cs107_package.subpkg_1.module_2.bar()"

def example_function():
    """If you have code that raises exceptions, pytest can verify them."""
    raise RuntimeError("This function should not be called")

def test_example_function():
    with pytest.raises(RuntimeError):
        example_function()
```

pytest

And add the new test module in: `./tests/run_tests.sh`

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

# list of test cases you want to run
tests=(
    subpkg_1/test_module_1.py
    subpkg_1/test_module_2.py
)
```

Running the tests with our driver script:

```
$ ./run_tests.sh pytest -v
============================ test session starts ============================
platform linux -- Python 3.9.7, pytest-6.2.5, py-1.10.0, pluggy-0.13.1
cachedir: .pytest_cache
rootdir: /home/fabs/harvard/CS107/cs107_project/tests
plugins: cov-2.12.1, anyio-3.3.2
collected 4 items

subpkg_1/test_module_1.py::TestTypes::test_class_Foo PASSED           [ 25%]
subpkg_1/test_module_1.py::TestFunctions::test_function_foo PASSED    [ 50%]
subpkg_1/test_module_2.py::TestFunctions::test_bar PASSED             [ 75%]
subpkg_1/test_module_2.py::test_example_function PASSED               [100%]
============================= 4 passed in 0.02s =============================
```

Note: the new test module that we just created is designed for `pytest`. Running `./run_tests.sh` (defaults to `python -m unittest`) will only run 2 out of the 4 total tests. If you combine `unittest`'s and `pytest`'s, always run them with `pytest`.

`doctest`

- The `unittest` and `pytest` packages are the ones you should build your tests upon.
- `doctest`'s are small scale tests that you can integrate in the `docstring`'s of your `python` code.
- They are useful for providing examples in your documentation and serve as a conceptual test at the same time.
- A `doctest` can not accurately capture all corner cases without cluttering your documentation. Use them appropriately to indicate use cases and adhere to `unittest` and/or `pytest` for proper test suites.

doctest

Example: assume we have this content in `./src/cs107_package/subpkg_2/module_3.py`

```python
"""
This is the docstring for ./subpkg_2/module_3.py.  This module provides one
function `baz`.  Example usage is:

>>> baz(0)
0
"""

def baz(x):
    """
    Return the input x if it is an int or float.

    Arguments:
    x : input argument

    Returns:
    x if it is of type int or float

    Examples:
    >>> baz(0)
    0
    >>> baz(0.0)
    0.0
    >>> baz('a string')
    Traceback (most recent call last):
        ...
    ValueError: x must be int or float
    """
    if not isinstance(x, (int, float)):
        raise ValueError('x must be int or float')
    return x
```

doctest

Example: assume we have this content in `./src/cs107_package/subpkg_2/module_3.py`

```python
"""
This is the docstring for ./subpkg_2/module_3.py.  This module provides one
function `baz`.  Example usage is:

>>> baz(0)
0
"""

def baz(x):
    """
    Return the input x if it is an int or float.
    """
    if not isinstance(x, (int, float)):
        raise ValueError('x must be int or float')
    return x
```

- You can run the `doctest` for this module using:

  `python -m doctest [-v] src/cs107_package/subpkg_2/module_3.py`

- Or you can use `pytest` and let it auto discover:

  `pytest --doctest-modules [-v] src/`

ASSESSING THE QUALITY OF TESTS

- Once you have written tests, how sure can you be that your tests *cover* all the source lines of code (SLOC) in your code base?
- *Code coverage* (or test coverage) is a metric that expresses how much of your code base is executed by running your test suite(s).
- The metric usually expresses a *percentage* of covered code based on:
  - Function/method coverage: has each function or method in the program been called?
  - Line coverage: has each SLOC in the program been executed?
  - Branch coverage: has each branch path in the program been executed?

  Often *line coverage* is the most interesting.
- Code coverage tools simply generate the data that is then converted into a readable format like HTML or command line output.
- Code coverage can easily be integrated in the CI pipeline where each build generates data that can be uploaded to a service to track the history of code coverage.

CODE COVERAGE IN `python`

- Generating coverage reports in `python` is easy.
- Two prominent tools for this task are
  1. `coverage` (https://pypi.org/project/coverage/)
  2. `pytest-cov` (https://pypi.org/project/pytest-cov/ a plugin for `pytest`)
- Generating coverage reports involves the following steps:
  1. Instrumenting the code for coverage. If the program is compiled, a special binary is produced for this task.
  2. Running the test suites with the instrumented code/binary. This will result in a database of raw coverage data.
  3. Post-processing of the database allows to extract several statistics and reports.

CODE COVERAGE IN python

- We use pytest here, examples for coverage can be found at https://coverage.readthedocs.io/en/6.0.1/
- We can compute the coverage of our package with

```
$ pytest --cov=cs107_package --cov-report=term-missing
```

where the --cov argument specifies the python package we want to cover and it should report the lines that we do not cover.

- For this to work you must make sure PYTHONPATH is set correctly. As we already did this in our test driver we can just wrap around it:

```
$ ./run_tests.sh pytest --cov=cs107_package --cov-report=term-missing
============================ test session starts ==================================
----------- coverage: platform linux, python 3.9.7-final-0 -----------------------
Name
                                                        Stmts   Miss  Cover   Missing
-----------------------------------------------------------------------------------
cs107_project/src/cs107_package/__init__.py                 3      0   100%
cs107_project/src/cs107_package/__main__.py                 3      3     0%   1-4
cs107_project/src/cs107_package/subpkg_1/__init__.py        3      0   100%
cs107_project/src/cs107_package/subpkg_1/module_1.py        6      0   100%
cs107_project/src/cs107_package/subpkg_1/module_2.py        4      0   100%
cs107_project/src/cs107_package/subpkg_2/__init__.py        2      0   100%
cs107_project/src/cs107_package/subpkg_2/module_3.py        4      1    75%   31
cs107_project/src/cs107_package/subpkg_2/module_4.py        1      1     0%   1
cs107_project/src/cs107_package/subpkg_2/module_5.py        1      1     0%   1
-----------------------------------------------------------------------------------
TOTAL
                                                           27      6    78%
============================= 4 passed in 0.07s ===================================
```

CODE COVERAGE IN python

- To integrate code coverage in our CI pipeline, we can simply extend our `.travis.yml` file to run our tests with coverage enabled:

```
# These packages we only want in our CI builds
before_install: python -m pip install build pytest pytest-cov

# Run the tests defined in the package
script: (cd tests && ./run_tests.sh pytest --cov=cs107_package --cov-report=xml)
```

> ***Note:*** we now also need the `pytest` and `pytest-cov` packages in our build environment. We also write the report to a `xml` file instead of `stdout`.

- The data in the `xml` file can now be uploaded to a server that will keep track of the coverage history for the code base. This can be a self-hosted service or hosted services like coveralls.io or codecov.io (free for open-source projects).
- An example to upload our coverage reports to codecov.io from a TravisCI build could look like this in the `.travis.yml` file:

```
# Upload the coverage report to codecov
after_success:
  - curl -Os https://uploader.codecov.io/latest/linux/codecov
  - chmod +x codecov
  - ./codecov -t ${CODECOV_TOKEN}
```

The first line downloads the upload tool into the CI container (do not use the deprecated `bash` uploader!), the second line needs no explanation and the third line executes the upload tool using a secret token obtained from an environment variable.

CODE COVERAGE IN python

- An example to upload our coverage reports to codecov.io from a TravisCI build could look like this in the `.travis.yml` file:

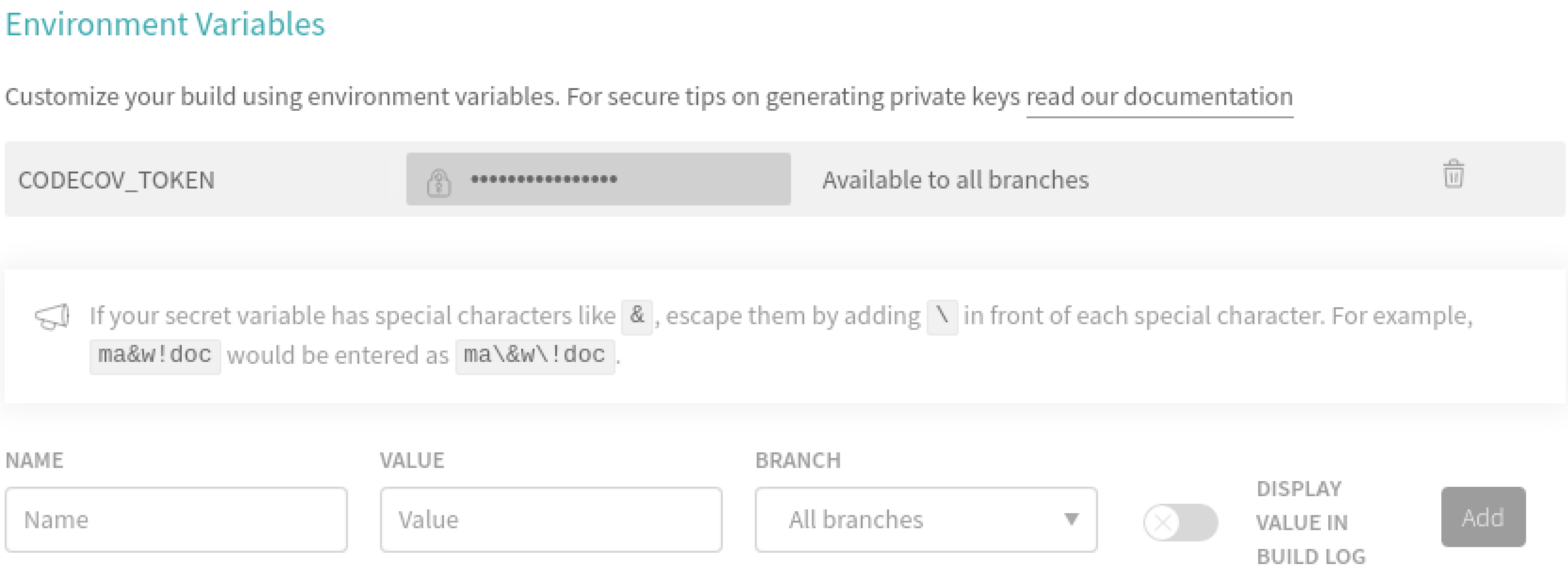
```
# Upload the coverage report to codecov
after_success:
  - curl -Os https://uploader.codecov.io/latest/linux/codecov
  - chmod +x codecov
  - ./codecov -t ${CODECOV_TOKEN}
```

  The first line downloads the upload tool into the CI container (do not use the deprecated `bash` uploader!), the second line needs no explanation and the third line executes the upload tool using a secret token obtained from an environment variable.

- The `CODECOV_TOKEN` is like a password, it tells codecov.io to which project this upload belongs. You should not expose such sensitive data in your `git` repository (even if it is private).

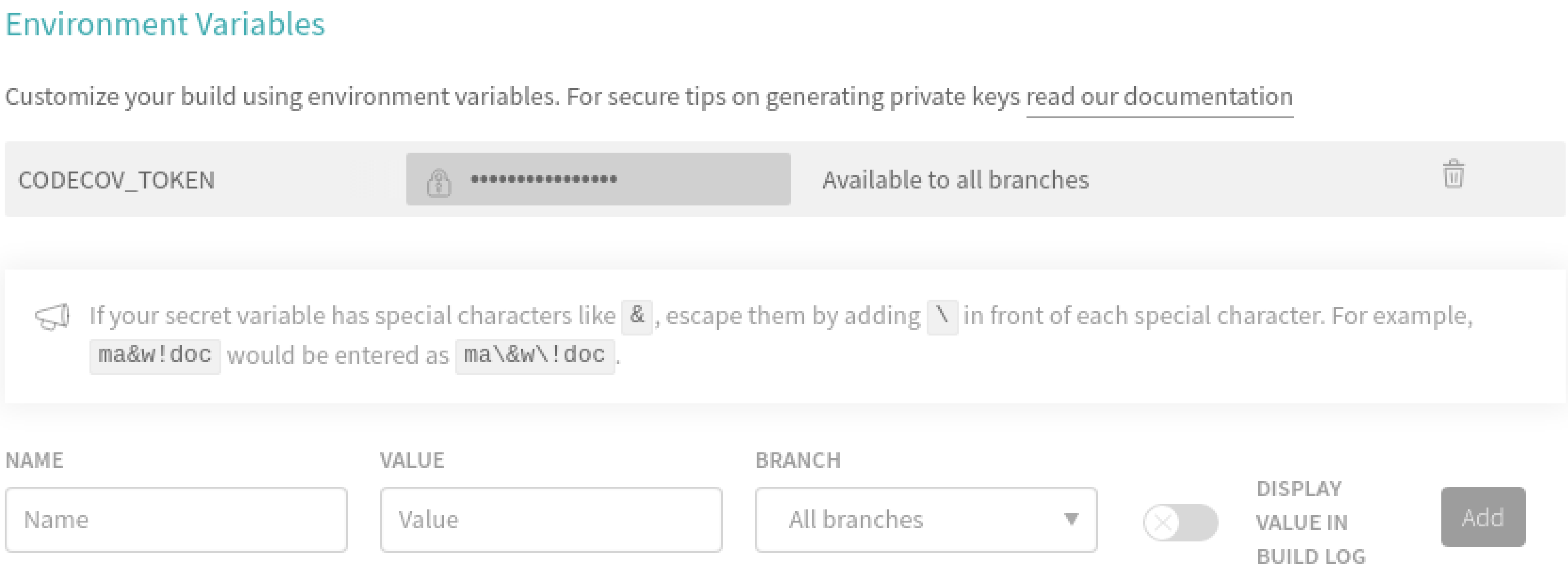
CODE COVERAGE IN python

- The `CODECOV_TOKEN` is like a password, it tells codecov.io to which project this upload belongs. You should not expose such sensitive data in your `git` repository (even if it is private).
- Define an environment variable in your TravisCI project settings instead:

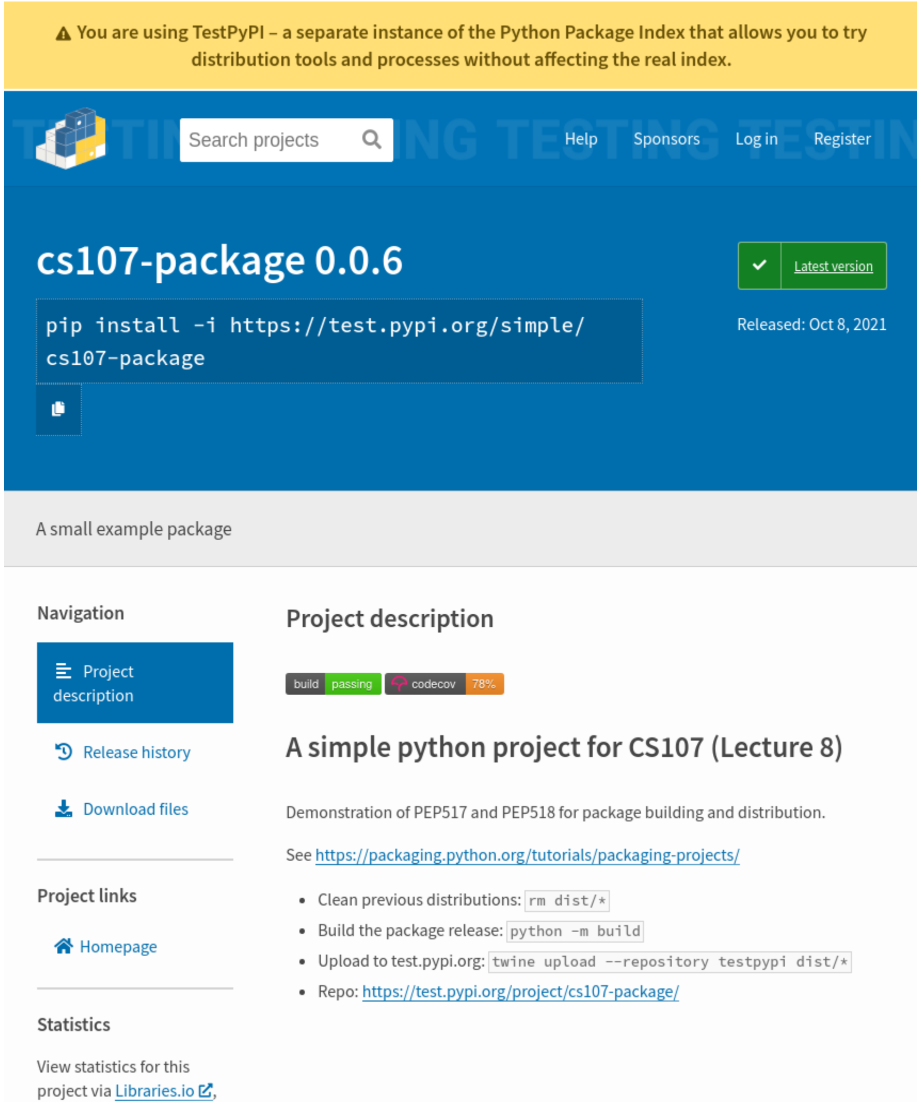


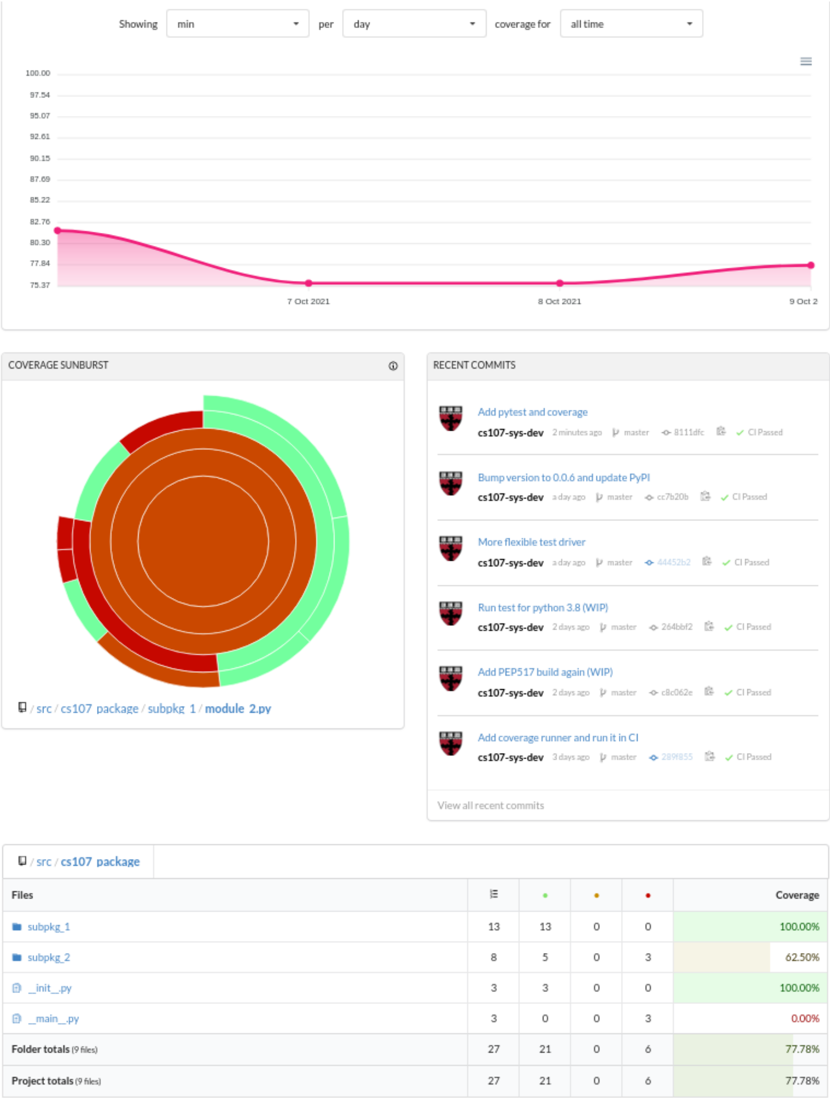
CODE COVERAGE IN `python`

- The `CODECOV_TOKEN` is like a password, it tells codecov.io to which project this upload belongs. You should not expose such sensitive data in your `git` repository (even if it is private).
- Define an environment variable in your TravisCI project settings instead:



CODE COVERAGE IN python





DOCUMENTATION

> *Finally, promotion of a program to a programming product requires its thorough documentation, so that anyone may use it, fix it, and extend it. As a rule of thumb, I estimate that a programming product costs at least three times as much as a debugged program with the same function.*
>
> Frederick Brooks, The Mythical Man-Month

DOCUMENTATION

- Documentation is an integral part of any software project and must follow the Software Requirements Specification (your contract with the customer).
- Once the Software Requirements Specification (SRS) is written and approved, the interfaces are defined and remain *invariant*.
- The *implementation* of such invariants (e.g. interfaces or other requirements in the SRS that must not change) is a detail and may change between different releases of the software.
- The best place to put documentation is right next to the code that it documents.
- We can document in two ways:
  1. By commenting code (intended for the developer/maintainer)
  2. In-source tools for documentation:
     - `python`: `docstrings` following PEP257, type hinting (since `python3.5`), sphinx
     - `C++`: doxygen, breathe

DOCUMENTATION: COMMENTS

Writing good comments is an art like writing good commit messages.

Following Jeff Atwood (founder of Stack Overflow):

1. The value of a comment is directly proportional to the distance between the comment and the code.
2. Comments with complex formatting cannot be trusted.
3. Don't include redundant information in the comments.
4. The best kind of comments are the ones you don't need.

The last item refers to *self-documenting* code. Attempt to write simple code where possible that can easily be understood by itself.

DOCUMENTATION: python DOCSTRINGS

The conventions for python docstrings are outlined in PEP257.

Example: numpy.dot

```
def dot(a, b, out=None):
    """
    dot(a, b, out=None)

    Dot product of two arrays.
    <skipping some documentation for brevity>

    Parameters
    ----------
    a : array_like
        First argument.
    b : array_like
        Second argument.
    out : ndarray, optional
        Output argument. This must have the exact kind that would be returned
        if it was not used.

    Returns
    -------
    output : ndarray
        Returns the dot product of `a` and `b`.  If `a` and `b` are both
        scalars or both 1-D arrays then a scalar is returned; otherwise
        an array is returned.

    Raises
    ------
    ValueError
        If the last dimension of `a` is not the same size as
        the second-to-last dimension of `b`.

    Examples
    --------
    >>> np.dot(3, 4)
    12

    Neither argument is complex-conjugated:

    >>> np.dot([2j, 3j], [2j, 3j])
    (-13+0j)
    """
```

- The documentation is very extensive and split into several *sections*. (Some content is stripped on the left.)
- The docstring for any object is stored in the `__doc__` attribute.
- python documentation is accessible through the pydoc module:

```
$ pydoc numpy.dot
Help on function dot in numpy:

numpy.dot = dot(...)
    dot(a, b, out=None)
...
```

or in code via

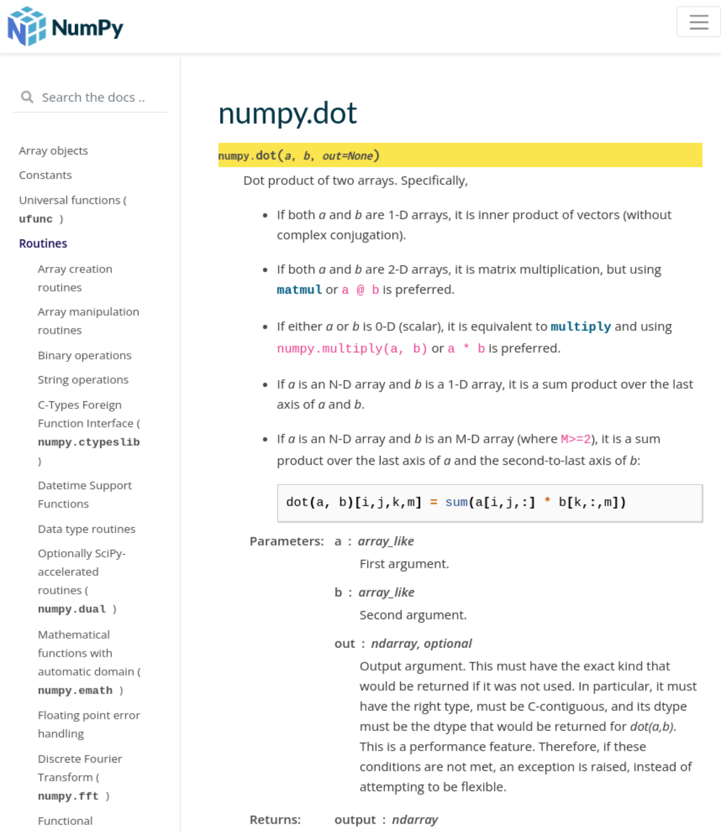
```
import pydoc
import numpy
pydoc.doc(numpy.dot)
```

DOCUMENTATION: python DOCSTRINGS

Once you have written the documentation of your code, you can use `sphinx` to generate online documentation for your project, for example on readthedocs.org.

Example: `numpy.dot`

```
def dot(a, b, out=None):
    """
    dot(a, b, out=None)

    Dot product of two arrays. Specifically,

    - If both `a` and `b` are 1-D arrays, it is inner product of vectors
      (without complex conjugation).

    - If both `a` and `b` are 2-D arrays, it is matrix multiplication,
      but using :func:`matmul` or ``a @ b`` is preferred.

    - If either `a` or `b` is 0-D (scalar), it is equivalent to :func:`multiply
      and using ``numpy.multiply(a, b)`` or ``a * b`` is preferred.

    - If `a` is an N-D array and `b` is a 1-D array, it is a sum product over
      the last axis of `a` and `b`.

    - If `a` is an N-D array and `b` is an M-D array (where ``M>=2``), it is a
      sum product over the last axis of `a` and the second-to-last axis of `b`:

        dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])

    Parameters
    ----------
    a : array_like
        First argument.
    b : array_like
        Second argument.
    out : ndarray, optional
        Output argument. This must have the exact kind that would be returned
        if it was not used. In particular, it must have the right type, must be
        C-contiguous, and its dtype must be the dtype that would be returned
        for `dot(a,b)`. This is a performance feature. Therefore, ...
    """
```


NumPy

Search the docs ...

Array objects
Constants
Universal functions (ufunc)
Routines
Array creation routines
Array manipulation routines
Binary operations
String operations
C-Types Foreign Function Interface (numpy.ctypeslib)
Datetime Support Functions
Data type routines
Optionally SciPy-accelerated routines (numpy.dual)
Mathematical functions with automatic domain (numpy.emath)
Floating point error handling
Discrete Fourier Transform (numpy.fft)
Functional

numpy.dot

numpy.dot(*a*, *b*, *out=None*)

Dot product of two arrays. Specifically,

- If both *a* and *b* are 1-D arrays, it is inner product of vectors (without complex conjugation).
- If both *a* and *b* are 2-D arrays, it is matrix multiplication, but using `matmul` or `a @ b` is preferred.
- If either *a* or *b* is 0-D (scalar), it is equivalent to `multiply` and using `numpy.multiply(a, b)` or `a * b` is preferred.
- If *a* is an N-D array and *b* is a 1-D array, it is a sum product over the last axis of *a* and *b*.
- If *a* is an N-D array and *b* is an M-D array (where `M>=2`), it is a sum product over the last axis of *a* and the second-to-last axis of *b*:

```
dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])
```

Parameters: **a** : *array_like*
First argument.
b : *array_like*
Second argument.
out : *ndarray, optional*
Output argument. This must have the exact kind that would be returned if it was not used. In particular, it must have the right type, must be C-contiguous, and its dtype must be the dtype that would be returned for *dot(a,b)*. This is a performance feature. Therefore, if these conditions are not met, an exception is raised, instead of attempting to be flexible.

Returns: **output** : *ndarray*

RECAP

- Continuous Integration (CI) in Software Development
- Testing your code and verifying the *quality* of your tests
- Documentation