Testing & Code Analysis

Bálint Aradi

Course: Scientific Programming / Wissenchaftliches Programmieren (Python)

Bremen Center for Computational Materials Science BCCMS

Universität Bremen

https://www.bccms.uni-bremen.de/people/b-aradi/wissen-progr/python/2023

Outline

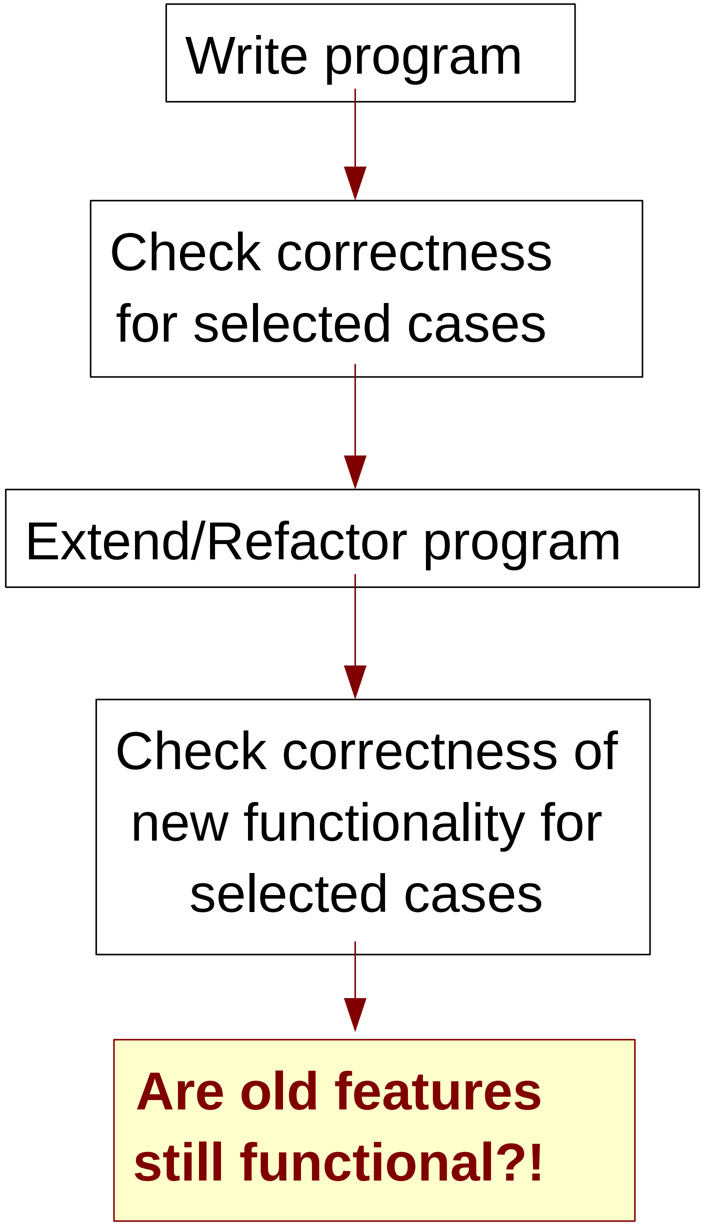
- Program testing (unit tests)
- Testing coverage
- Code quality analysis

You might need to install some Conda packages to try the examples in this lecture:

```
conda install pytest pytest-cov coverage pylint black
```

Program testing

Program testing



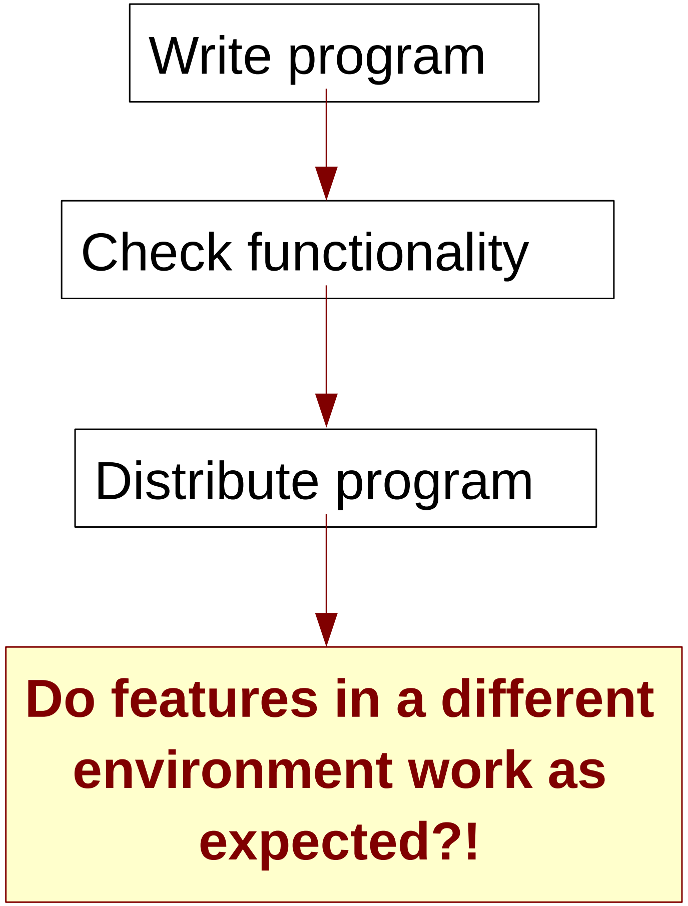
Write program

Check correctness for selected cases

Extend/Refactor program

Check correctness of new functionality for selected cases

Are old features still functional?!



Write program

Check functionality

Distribute program

Do features in a different environment work as expected?!

Program testing

When to test?

- Package functionality/integrity must be tested after each (relevant) change
- Package functionality/integrity must be tested whenever it is used in a different environment

How to test?

Effort needed to carry out tests must be as low as possible

- It should be possible to run all (or seleted tests) with one command
- Tests should be reasonably fast
- Correctness of the results should be checked automatically

Automated testing (with test protocol) is an essential part of the development

Testing during development

Unit tests – white box testing

- Each program unit (e.g. function) is tested independently
- Check whether for given input the right output is returned

Regression tests – black box testing

- Testing the package functionality as whole
- Tesintg whether for given input (e.g. user supplied data) expected output is generated
- Often includes stress-tests or scaling tests

Test driven development (e.g. agile programming)

- **First** write the tests for a given functionality, **then** imlement the functionality
- If a bug is found, add it as test first (improve **coverage**) and then fix it so that it passes the test

Automatic Python testing frameworks

Unittest package in Python

- Comes as package with the standard Python 3 distribution (out of the box)
- Powerful with a lot of features
- Requires object-oriented approach to define tests

[Unittest documentation]

Pytest package

- Third party package (extra dependency, although quite standard)
- Extremly powerful and versatile, actively developed with large community
- Works both, with procedure and object oriented approach
- Simple tests can be set up with a few lines of code

[Pytest documentation]

Writing simple tests in Pytest

mymath.py

```
import numpy as np


def factorial(nn):
    """Calculates the factorial of a number


    Args:
        nn: Number to calculate the factorial of.


    Returns:
        Factorial of the argument.
    """
    res = 1
    for ii in range(2, nn + 1):
      res *= ii
    return res
```

Writing simple tests in Pytest

1. Write functions for testing given procedures / functionality
2. Function should indicate test result (success / failure) using **assert**

test_mymath.py

```python
import mymath

def test_factorial_5():
    "Test 5!"
    result = mymath.factorial(5)
    assert result == 120

def test_factorial_0():
    "Test 0!"
    result = mymath.factorial(0)
    assert result == 1
```

The name of the test functions must start with **“test”**

assert: If expression evaluates to **False,** code execution is stopped (an exception is raised to signalize failure) otherwise execution is continued

Running the tests from the shell

- Go to directory with the test file
- Start Python and import the pytest module

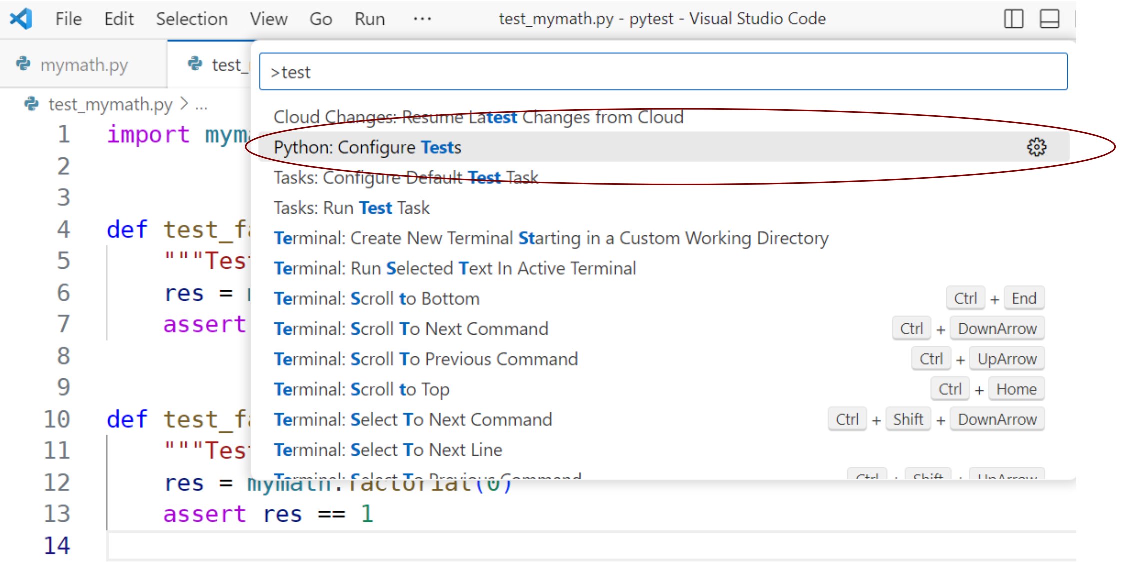
```
python3 -m pytest
```

```
python -m pytest
```

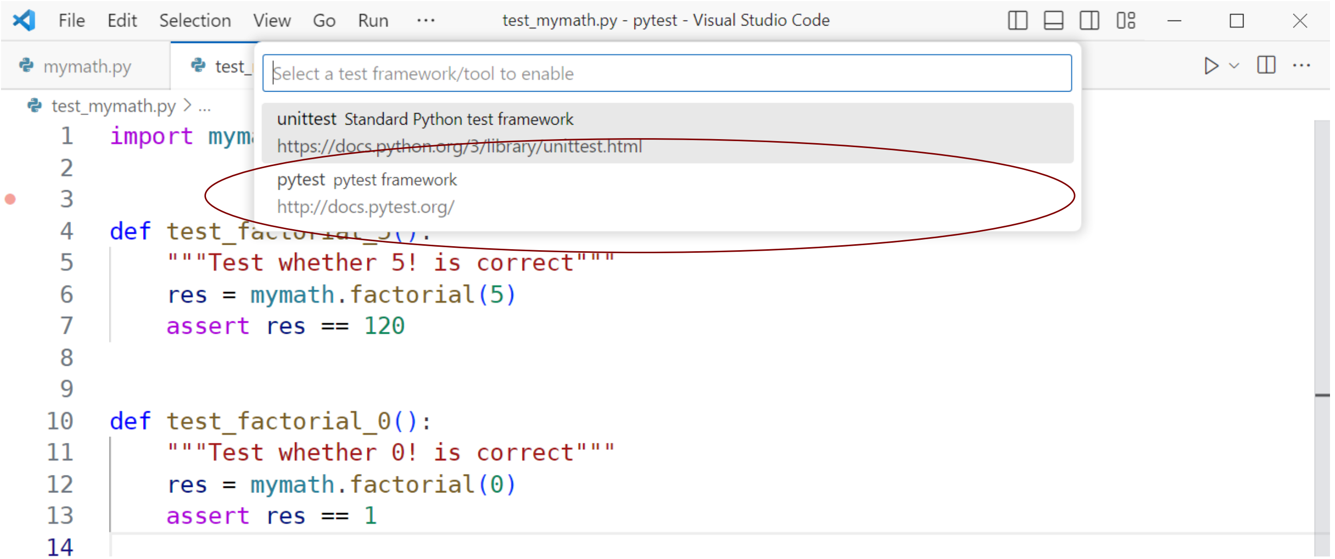
- When pytest is imported in a script, it will automatically start **test-discovery**
- It will scan all Python source files in the given directory for test functions and execute all tests found (all functions with names prefixed by “test”)

```
======== test session starts =======
test_mymath.py ...

====== 2 passed in 0.13 seconds ====
```

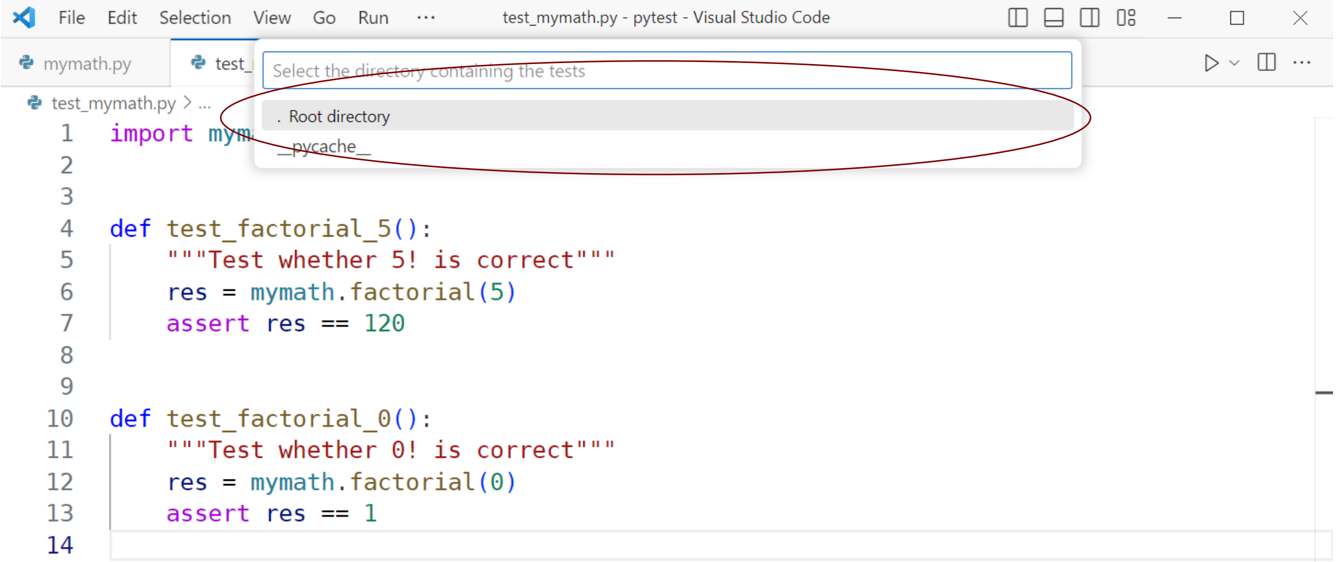
Running tests from VSCode


File Edit Selection View Go Run ···
test_mymath.py - pytest - Visual Studio Code
mymath.py
test_
test_mymath.py > ...
>test
Cloud Changes: Resume Latest Changes from Cloud
Python: Configure Tests
Tasks: Configure Default Test Task
Tasks: Run Test Task
Terminal: Create New Terminal Starting in a Custom Working Directory
Terminal: Run Selected Text In Active Terminal
Terminal: Scroll to Bottom Ctrl + End
Terminal: Scroll To Next Command Ctrl + DownArrow
Terminal: Scroll To Previous Command Ctrl + UpArrow
Terminal: Scroll to Top Ctrl + Home
Terminal: Select To Next Command Ctrl + Shift + DownArrow
Terminal: Select To Next Line

```
import mym


def test_f
    """Tes
    res = r
    assert


def test_f
    """Tes
    res = mymath.factorial(0)
    assert res == 1
```

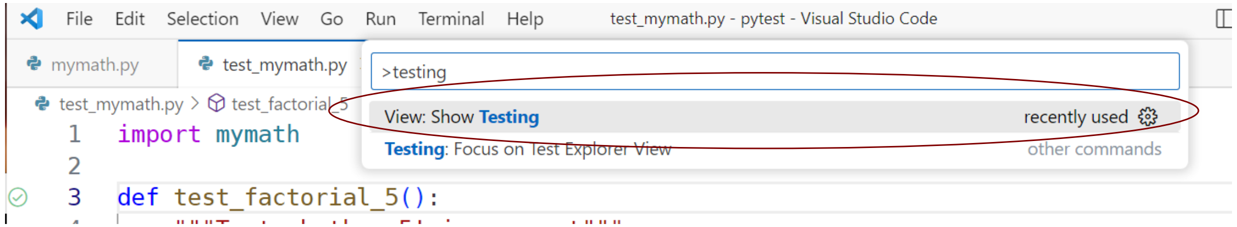

File Edit Selection View Go Run ···
test_mymath.py - pytest - Visual Studio Code
mymath.py
test_
test_mymath.py > ...
Select a test framework/tool to enable
unittest Standard Python test framework
https://docs.python.org/3/library/unittest.html
pytest pytest framework
http://docs.pytest.org/

```
import myma


def test_factorial_5():
    """Test whether 5! is correct"""
    res = mymath.factorial(5)
    assert res == 120


def test_factorial_0():
    """Test whether 0! is correct"""
    res = mymath.factorial(0)
    assert res == 1
```


File Edit Selection View Go Run ···
test_mymath.py - pytest - Visual Studio Code
mymath.py
test_
test_mymath.py > ...
Select the directory containing the tests
. Root directory
__pycache__

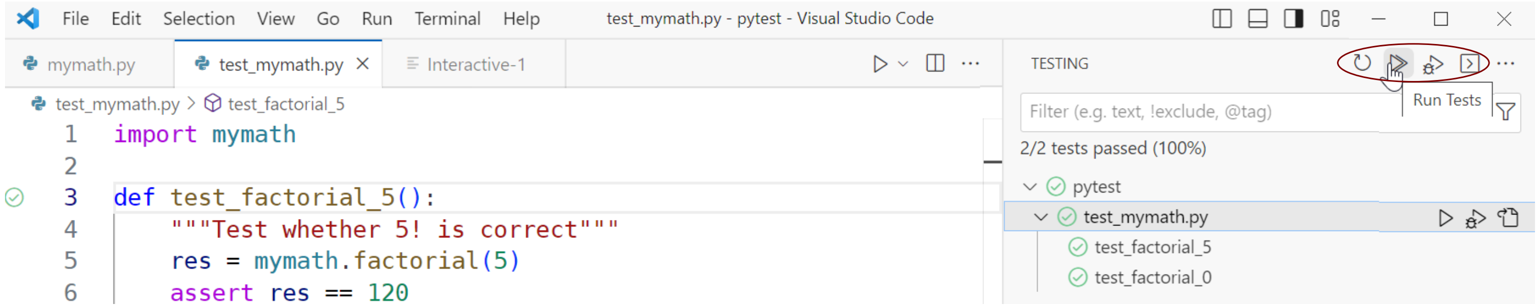
```
import mym


def test_factorial_5():
    """Test whether 5! is correct"""
    res = mymath.factorial(5)
    assert res == 120


def test_factorial_0():
    """Test whether 0! is correct"""
    res = mymath.factorial(0)
    assert res == 1
```

Running tests from VSCode


File Edit Selection View Go Run Terminal Help
test_mymath.py - pytest - Visual Studio Code
mymath.py
test_mymath.py
test_mymath.py > test_factorial_5
View: Show Testing
recently used
Testing: Focus on Test Explorer View
other commands

```
>testing
```

```python
import mymath

def test_factorial_5():
```

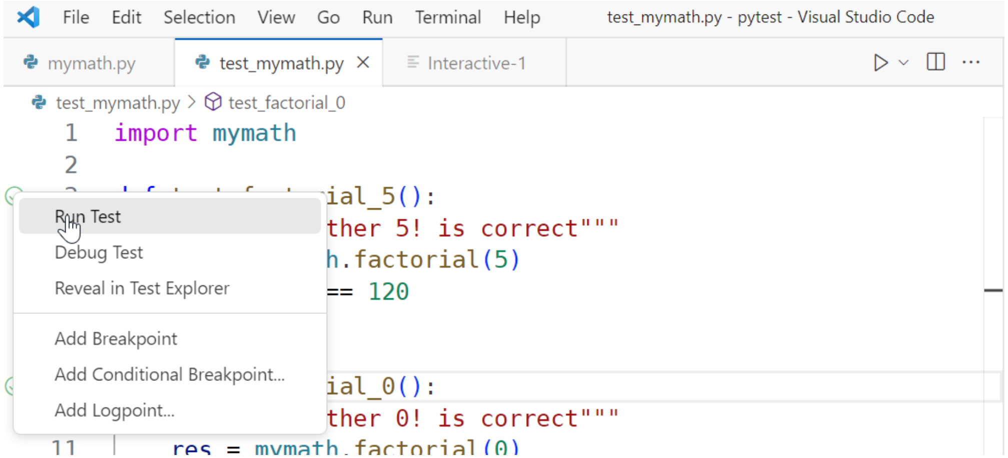

File Edit Selection View Go Run Terminal Help
test_mymath.py - pytest - Visual Studio Code
mymath.py
test_mymath.py
Interactive-1
test_mymath.py > test_factorial_5
TESTING
Run Tests
Filter (e.g. text, !exclude, @tag)

```python
import mymath

def test_factorial_5():
    """Test whether 5! is correct"""
    res = mymath.factorial(5)
    assert res == 120
```

```
2/2 tests passed (100%)
pytest
  test_mymath.py
    test_factorial_5
    test_factorial_0
```

Run all tests

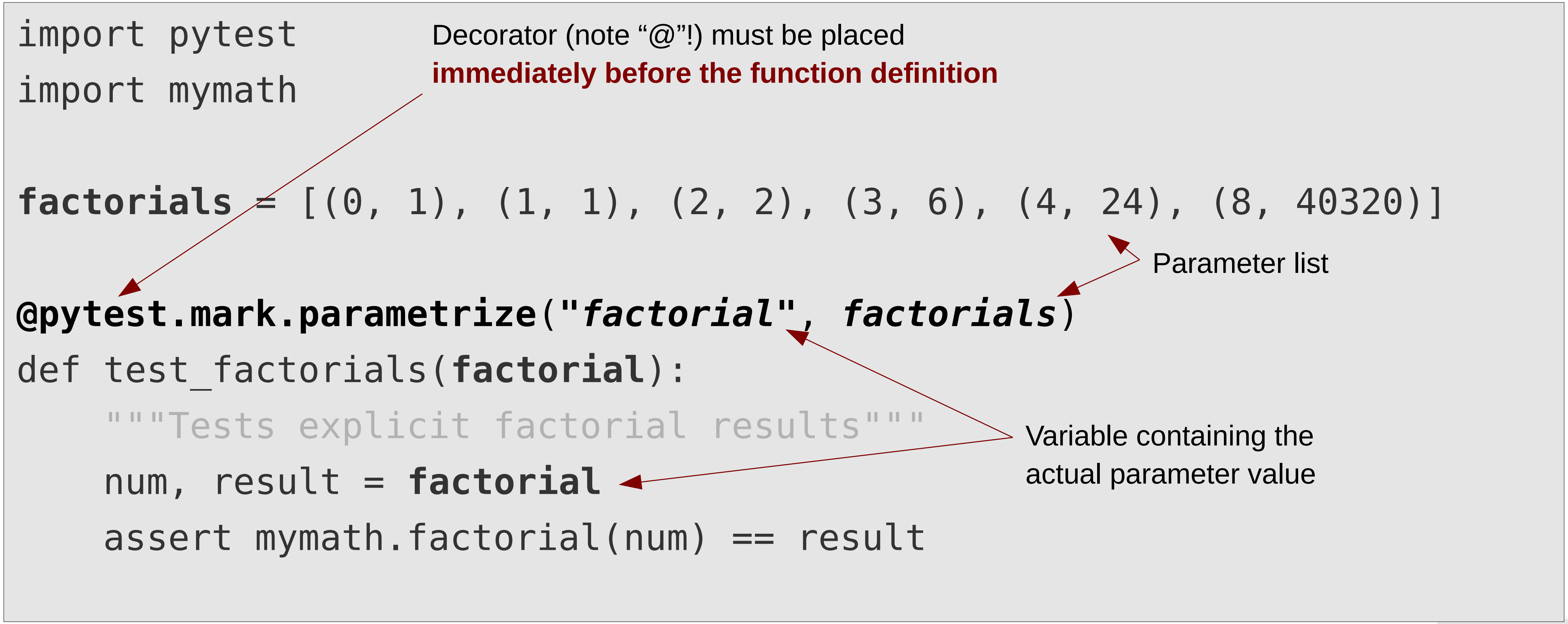
Run individual test
(right click)


File Edit Selection View Go Run Terminal Help
test_mymath.py - pytest - Visual Studio Code
mymath.py
test_mymath.py
Interactive-1
test_mymath.py > test_factorial_0
Run Test
Debug Test
Reveal in Test Explorer
Add Breakpoint
Add Conditional Breakpoint...
Add Logpoint...

```python
import mymath

                ial_5():
             ther 5! is correct"""
             h.factorial(5)
             == 120

             ial_0():
             ther 0! is correct"""
    res = mymath.factorial(0)
```

Parametrized tests

- When same test should be run several times with different input data
- **pytest.mark.parametrize** decorator executes test function for various tests by running over a list of parameters and passing one parameter at a time to the test function



```
import pytest
import mymath

factorials = [(0, 1), (1, 1), (2, 2), (3, 6), (4, 24), (8, 40320)]

@pytest.mark.parametrize("factorial", factorials)
def test_factorials(factorial):
    """Tests explicit factorial results"""
    num, result = factorial
    assert mymath.factorial(num) == result
```

Decorator (note “@”!) must be placed **immediately before the function definition**

Parameter list

Variable containing the actual parameter value

Parametrized tests

Example

- Prepare input and expected result (e.g. loading from disc)
- Calculate result using prepared input, compare result with prepared result

```python
import pytest
import solvers
TESTNAMES = ['simple', 'needs_pivot']

@pytest.mark.parametrize("testname", TESTNAMES)
def test_successful_elimination(testname):
    """Tests successful elimination."""
    aa, bb = get_test_input(testname)
    xx_expected = get_test_output(testname)
    xx_gauss = solvers.gaussian_eliminate(aa, bb)
    assert np.all(np.abs(xx_gauss - xx_expected) < 1e-10)
```

Decorator must be placed **immediately before function definition**

Test fixture

- When multiple tests need the same initialization
- **@pytest.fixture** decorator defines an initialization function
- Return value of fixture function is passed to tests with appropriate argument
- Fixture function is called for each test separately

```
import numpy.random as random
import pytest
import mymath

@pytest.fixture
def smallrandint():
    """Returns random integer from interval [1, 10]
    rand = random.random()
    randint = int(10 * rand) + 1
    return randint
```

Fixture function

Result returned by fixture function will be used in the appropriate tests

Test fixture

```
def test_lower_consistency(smallrandint):
    """Consistency with lower factorial"""
    nn = smallrandint
    factn = mymath.factorial(nn)
    assert factn == nn * mymath.factorial(nn - 1)

def test_upper_consistency(smallrandint):
    """For consistency with upper factorial"""
    nn = smallrandint
    factn = mymath.factorial(nn)
    assert mymath.factorial(nn + 1) == (nn + 1) * factn
```

Calls fixture **smallrandit()** and initializes argument with its return value

Argument name must match fixture function name

Useful functions when comparing arrays

- When two arrays (or an array and an integer) are compared, the comparison is made elementwise
- Result: array of logicals with the results of each elementwise comparison

```
aa = np.array([1, -2, 9])
aa < 0
```

→

```
[False  True False]
```

np.any() Checks whether any elements of an array evaluate to **True**

```
np.any(aa < 0)
```

→

```
True
```

np.all() Checks whether all elements of an array evaluate to **True**

```
np.all(aa < 0)
```

→

```
False
```

np.where() Returns elementwise 2nd or 3rd argument depending on logical values in 1st

```
np.where(aa < 0, 0, aa)
```

→

```
[1, 0, 9]
```

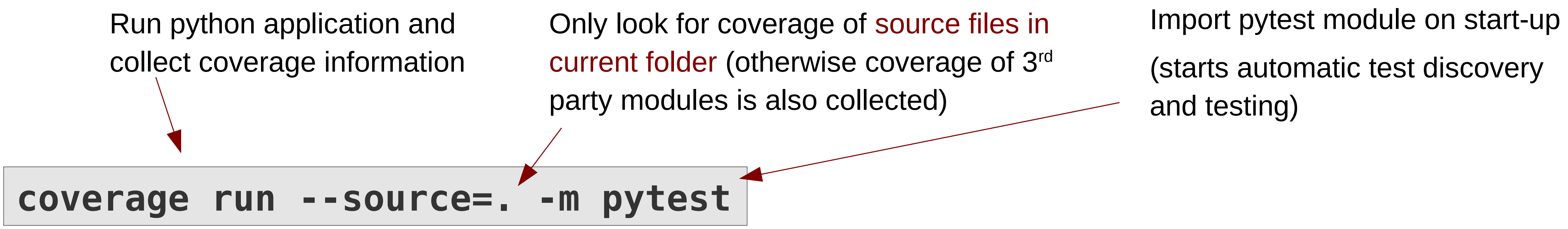
Test coverage

Test coverage

- Indicates which amount of the total code lines have been executed at least ones during the tests.
- Desirable: 100%
- Note: 100% coverage does not mean bug free code!
  It only means, that each line has been reached at least once during some tests. The code still can misbehave, if given line is executed with different (non-tested) data.

Collect coverage data

- **coverage** can collect coverage data while running a Python application
- It can be used together with Pytest to collect coverage info during testing (provided the coverage plugin for Pytest is installed)



Run python application and collect coverage information

Only look for coverage of source files in current folder (otherwise coverage of 3rd party modules is also collected)

Import pytest module on start-up

(starts automatic test discovery and testing)

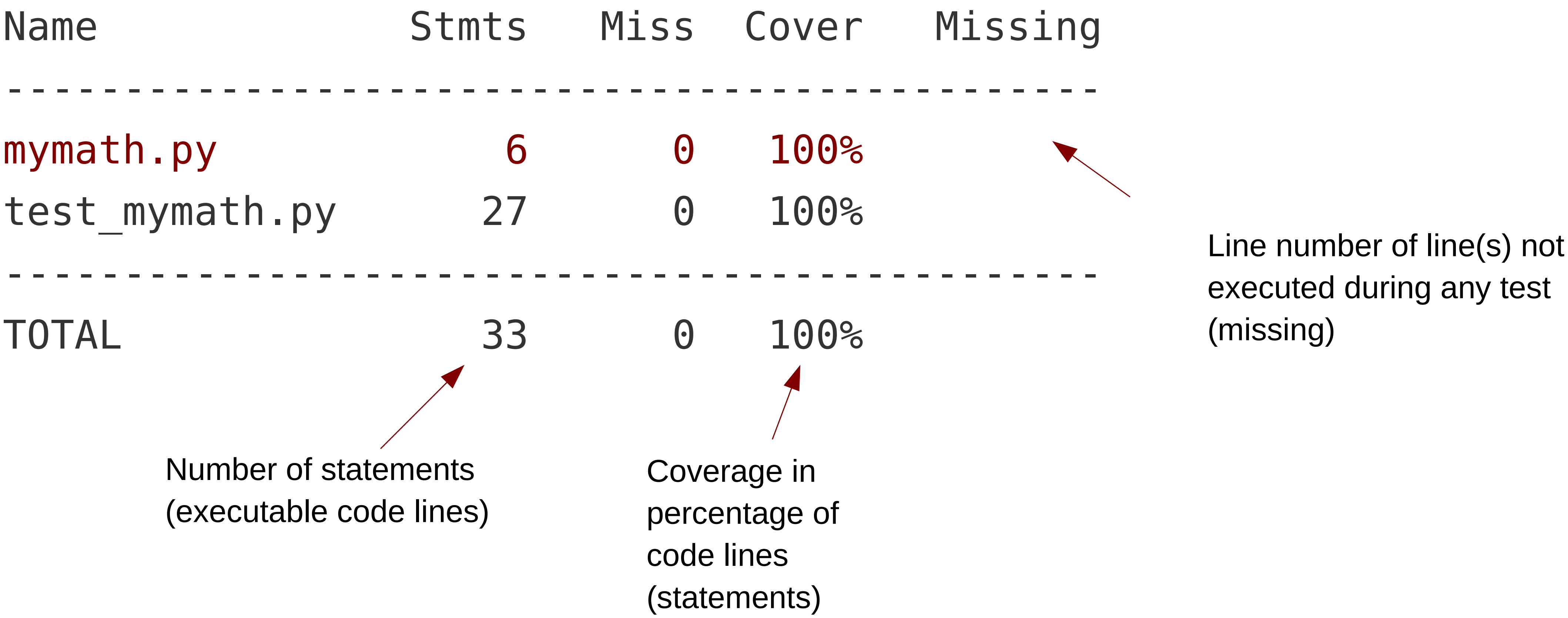
```
coverage run --source=. -m pytest
```

```
============================ test session starts ...
platform linux -- Python 3.5.2, ...
rootdir: /home/aradi/pyprojects/linsolver, inifile:
plugins: cov-2.2.1
collected 10 items

test_mymath.py ..........
```

Visualize coverage data

Short summary on the console

```
coverage report -m
```



```
Name              Stmts   Miss  Cover   Missing
-----------------------------------------------
mymath.py             6      0   100%
test_mymath.py       27      0   100%
-----------------------------------------------
TOTAL                33      0   100%
```

Line number of line(s) not executed during any test (missing)

Number of statements (executable code lines)

Coverage in percentage of code lines (statements)

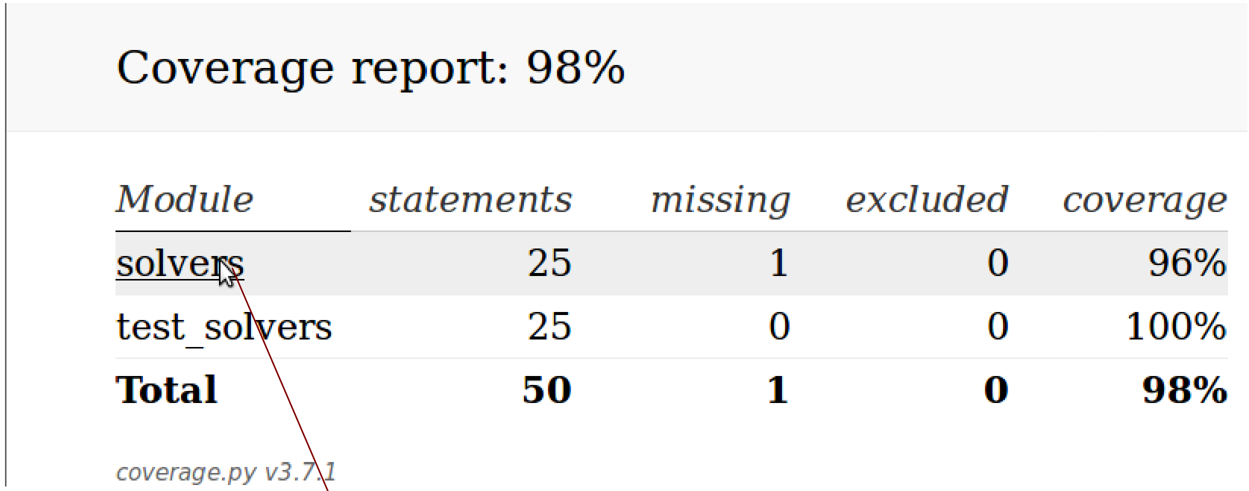
Visualize coverage data

Detailed coverage information in HTML

```
coverage html -d coverage_html
```

Directory where HTML pages should be stored

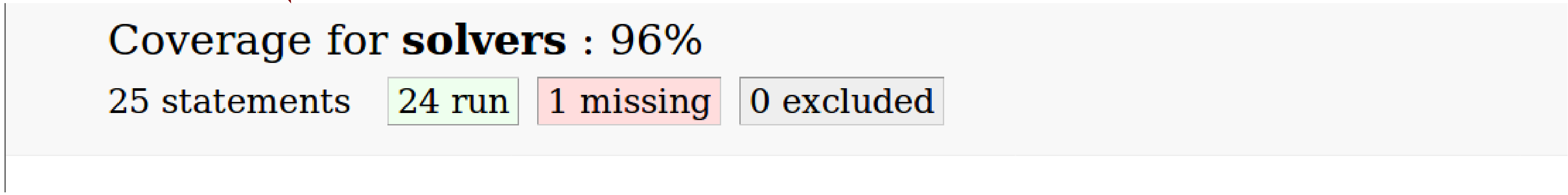
Open coverage_html/index.html in a browser



Coverage report: 98%

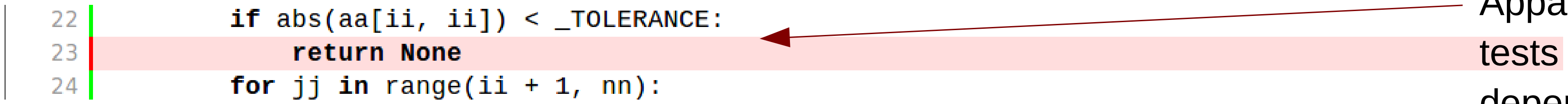
Module	*statements*	*missing*	*excluded*	*coverage*
solvers	25	1	0	96%
test_solvers	25	0	0	100%
Total	**50**	**1**	**0**	**98%**

coverage.py v3.7.1



Coverage for **solvers** : 96%

25 statements 24 run 1 missing 0 excluded



```
        if abs(aa[ii, ii]) < _TOLERANCE:
            return None
        for jj in range(ii + 1, nn):
```

Apparently none of the tests contained a linearly dependent system of equations …

Code quality analysis

Code analysis with Pylint

- Pylint reads Python source files and checks for possible convention breaches, inconsistencies and errors
- It produces a score for “code quality” (how much the code aligns to pylints guidelines)

Running pylint from command line

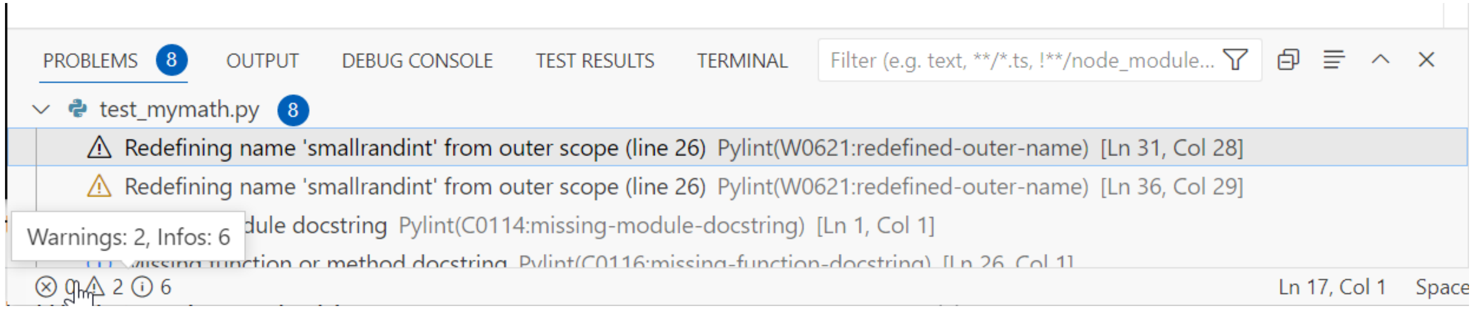
- Pass file name to the pylint program

```
pylint mymath.py
```

Running pylint from VS Code

- Install Pylint extension (if not installed yet)
- Select Pylint as Linter in the comman palette:

```
> Python: Select Linter
```



```
⚠ Redefining name 'smallrandint' from outer scope (line 26)  Pylint(W0621:redefined-outer-name)  [Ln 31, Col 28]
⚠ Redefining name 'smallrandint' from outer scope (line 26)  Pylint(W0621:redefined-outer-name)  [Ln 36, Col 29]
  dule docstring  Pylint(C0114:missing-module-docstring)  [Ln 1, Col 1]
  missing function or method docstring  Pylint(C0116:missing-function-docstring)  [Ln 26, Col 1]
```

- Linting on the fly ...

Configuring pylint

- Pylint reads the **~/.pylintrc** configuration file, if present
- Behaviour of pylint can be customized globally through the config file

Some customization suggestions

- Let pylint enable variable names with two letters
- Disable call check for numpy functions and classes
  (pylint often does fails to find the definitions in the numpy module)

Download the pylint configuration file from the course website and store it as **~/.pylintrc**

Disabling a check locally (for a file or a line)

- You can disable a given check locally by special comments:

```
# pylint: disable=W0621
```

Disables warning W0621 for the given file/line containing the comment

Python coding standard (PEP 8) & Black formatter

PEP 8 coding standard

- Python has a widely accepted coding style guide
- It has been documented in the **Python Enhancement Proposal 8** (PEP 8)
- Most Python projects stick to that standard
- Do not deviate from it without very-very good reasons

Black formatter

- Reformats code to be PEP 8 compatible (and makes some stylistic choices)

```
black  mymath.py
```

```
reformatted mymath.py
All done!
1 file reformatted.
```

- Best time for reformatting: Before adding the file to the stage (git add)

Have fun!