

7 – Testing & Code Analysis

Bálint Aradi

Course: Scientific Programming / Wissenschaftliches Programmieren (Python)



Installing necessary components

We will need additional programs:

- Pytest
- Pylint3
- Pyflakes3
- Pytest coverage plugin

```
sudo apt install python3-pytest pylint3 pyflakes3  
python3-pytest-cov python3-spyder-unittest
```

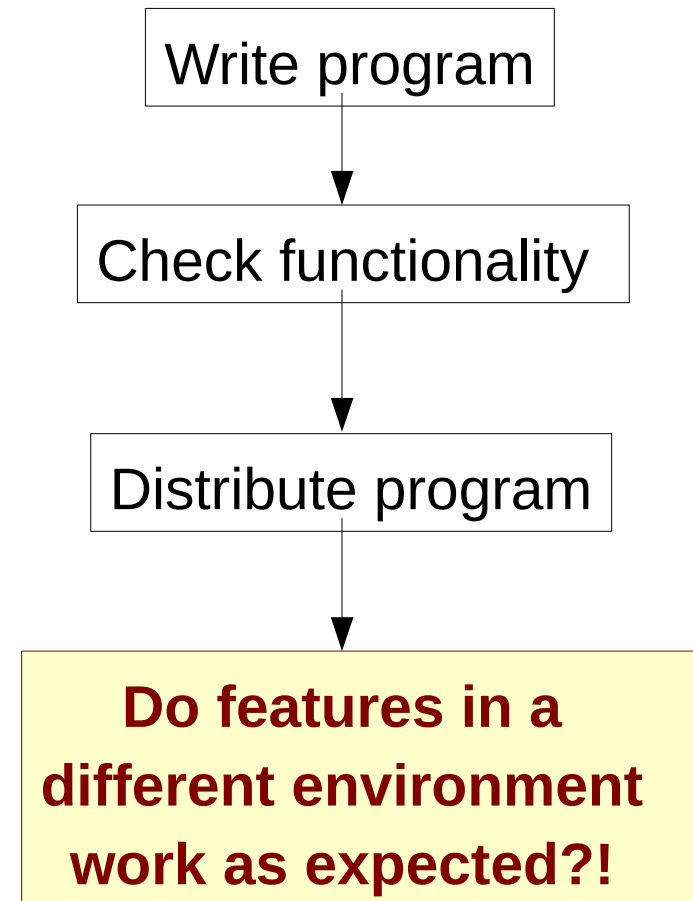
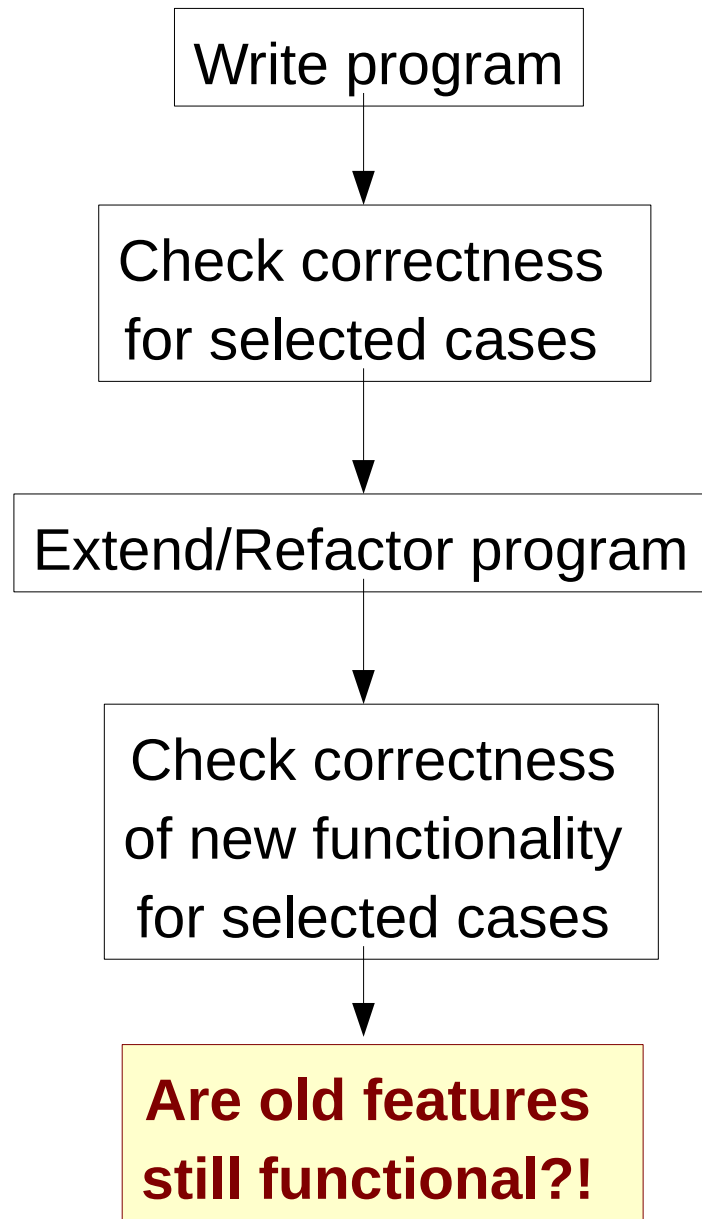
Outline

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



Program testing

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 selected 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
- Testing 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** implement the functionality
- If a bug is found, add it as test first (improve **coverage**) and then fix it so that it passes the test

Unittest package in Python (a.k.a. unittest2)

[\[Unittest documentation\]](#)

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

Pytest package

[\[Pytest documentation\]](#)

- Third party package (extra dependency, although quite standard)
- Extremely 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

Nose / Nose 2 package

[\[Nose 2 project site\]](#)

- Third party package (extra dependency)
- In many respects similar to Pytest
- Small community, future development rather unsure

Writing simple tests in Pytest

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

```
import mymath ← Assume we have defined a factorial() function in the mymath module

def test_factorial_small():
    "Test the factorial function for a small number"
    result = mymath.factorial(5)
    assert result == 120

def test_factorial_zero():
    "Test whether the factorial of zero is correct"
    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 tests from the shell

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

```
python3 -m pytest
```

- When pytest is imported in an executed 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”)

```
python3 -m pytest
```

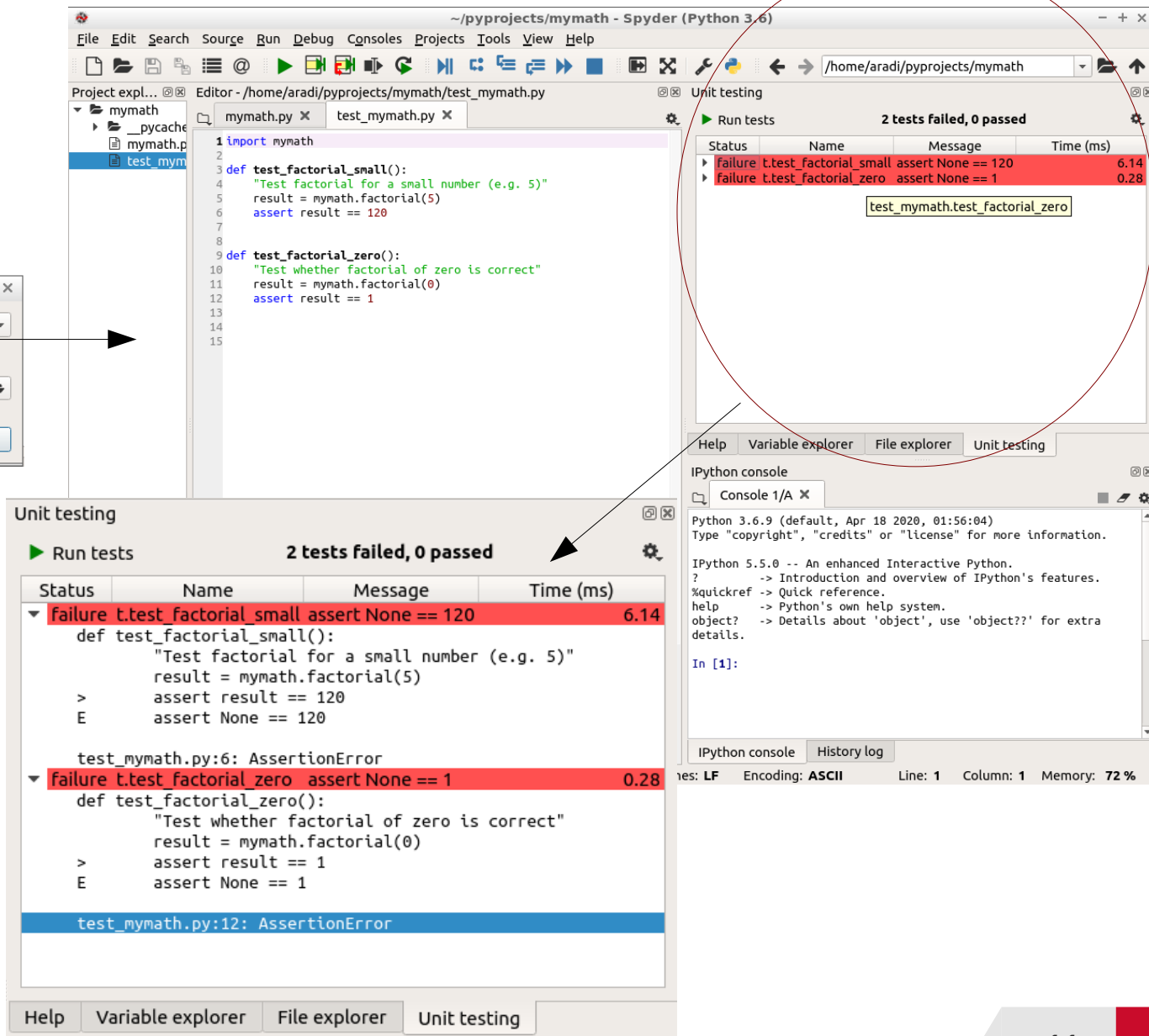
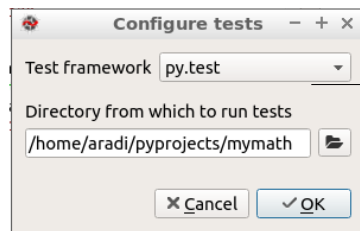
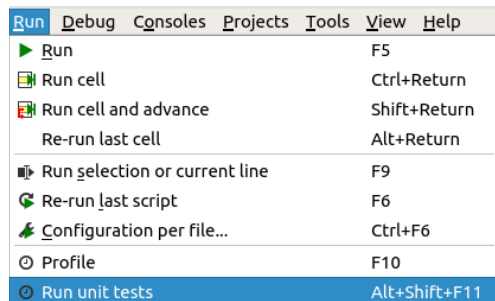
```
===== test session starts ...
```

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

```
===== 3 passed in 0.13 seconds ...
```

Running tests from Spyder

- Use the unittest extension of Spyder to run the unit tests:



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
```

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

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

Parameter list

```
@pytest.mark.parametrize("factorial", factorials)
```

```
def test_factorials(factorial):
```

```
    "Tests explicit factorial results"
```

```
    fact, result = factorial
```

```
    assert mymath.factorial(fact) == result
```

Variable containing the
actual parameter value

Parametrized tests

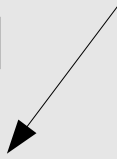
Example

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

```
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():
    rand = random.random()
    # Transforming integer into range [1, 10]
    randint = int(10 * rand) + 1
    return randint
```

Fixture function

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

Test fixture

```
@pytest.fixture
```

```
def smallrandint():
```

```
    ... # See previous slide
```

```
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"
```

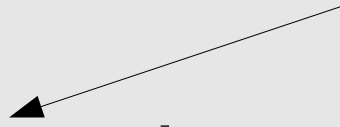
```
    nn = smallrandint
```

```
    factn = mymath.factorial(nn)
```

```
    assert mymath.factorial(nn + 1) == (nn + 1) * factn
```

Calls fixture **smallrandint()** 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 (**True** – 2nd, **False** – 3rd)

```
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

- **python3-coverage** (sometimes only **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)

```
python3-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 2 items
```

```
test_solvers.py ..
```

Import pytest module on start-up
(starts automatic test discovery
and testing)

Visualize coverage data

Short summary on the console

```
python3-coverage report -m
```

Name	Stmts	Miss	Cover	Missing

solvers	25	1	96%	23
test_solvers	25	0	100%	

TOTAL	50	1	98%	

Number of statements
(executable code lines)

Coverage in
percentage of
code lines
(statements)

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

Visualize coverage data

Detailed coverage information in HTML

```
python3-coverage html -d coverage_html  
firefox coverage_html/index.html
```

Directory where
HTML pages
should be stored

Coverage report: 98%

Module	statements	missing	excluded	coverage
<u>solvers</u>	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

```
22 |         if abs(aa[ii, ii]) < _TOLERANCE:  
23 |             return None  
24 |         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)

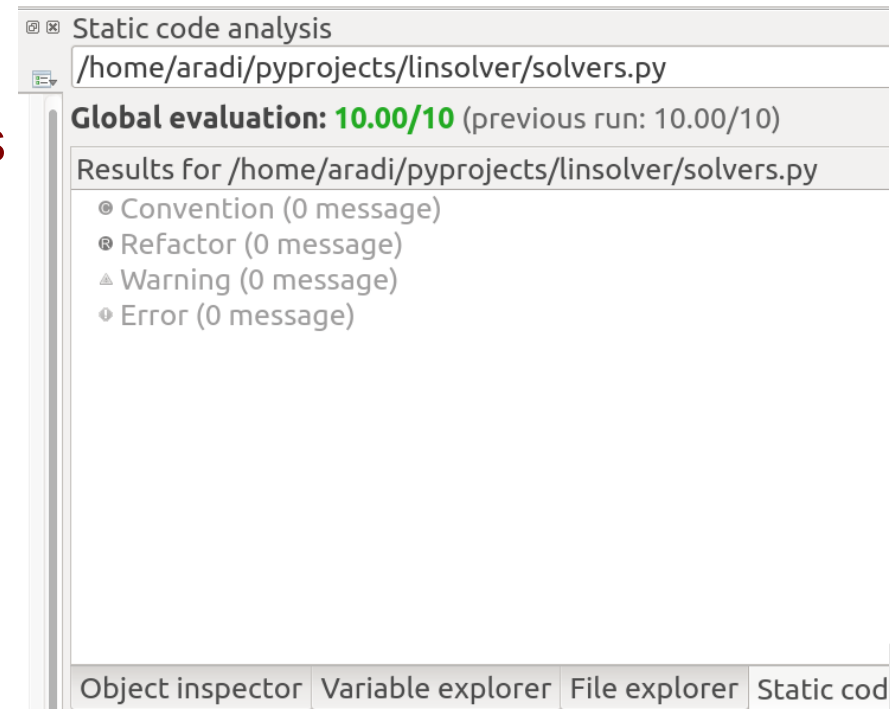
Running pylint from command line

- Pass file name to the pylint program

```
pylint3 solvers.py
```

Running pylint from Spyder

- Pressing **F8** activates **Pylint analysis**
- Results appear in the right upper window
- By clicking on the list items, the corresponding line is shown in the editor



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:

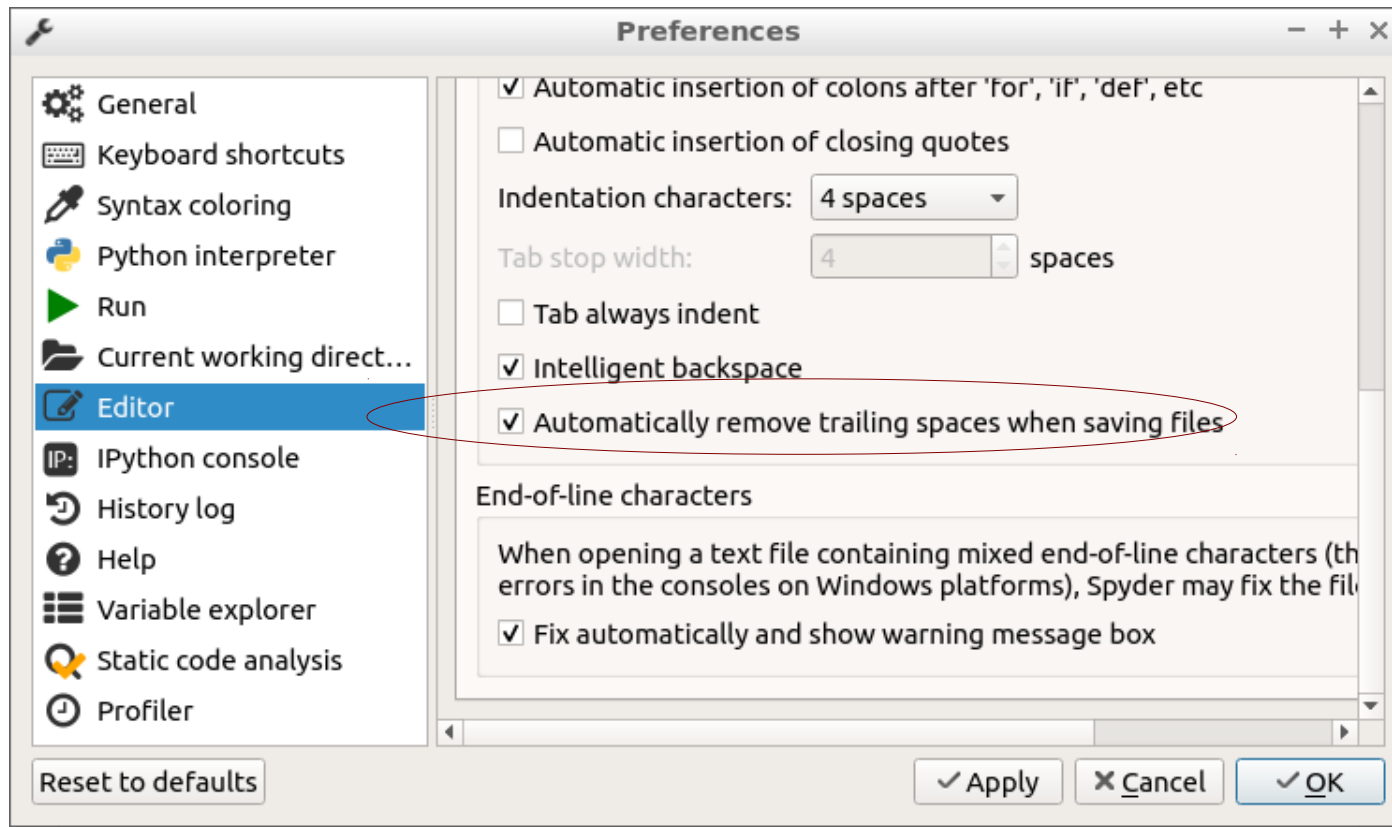
```
# pylint: disable=W0621
```

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

Configuring Spyder

- Set up Spyder to **remove trailing spaces** automatically (to avoid lots of Pylint warnings about convention breaches)

Tools / Preferences / Editor / Advanced settings

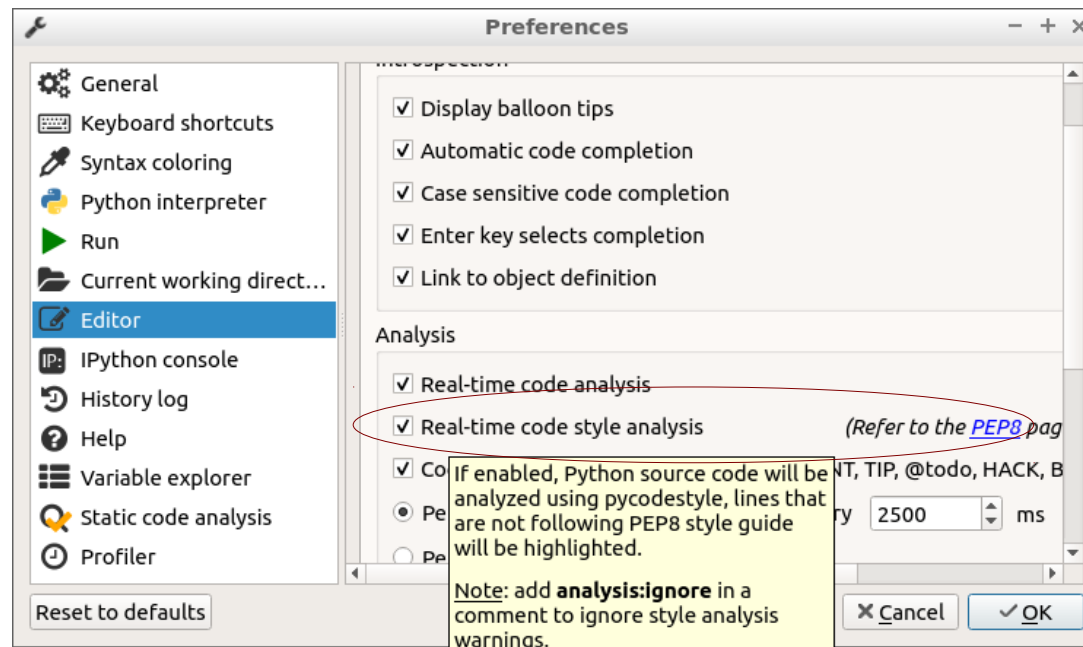


Only use this, if all participant of your project have set up their editors similarly!

Python coding standard (PEP8)

- 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
- **Spyder** can set up to **check for PEP 8 conformance** (recommended)

Tools / Preferences / Code Introspection



Set up
Spyder to
check
for PEP 8
compliance!