

# 15 – Unit testing

Bálint Aradi

**Scientific Programming in Python (2026)**

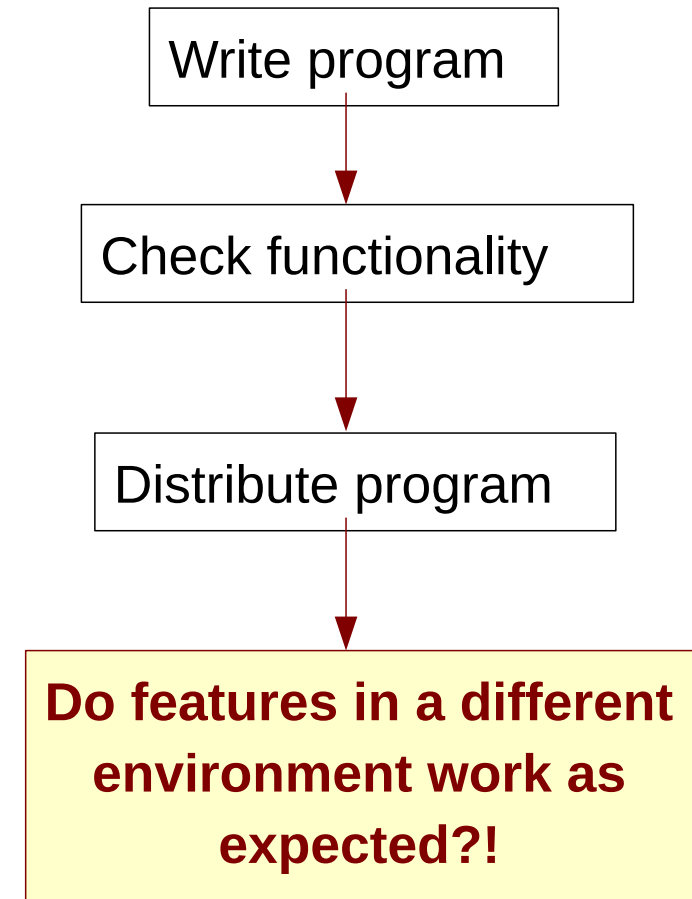
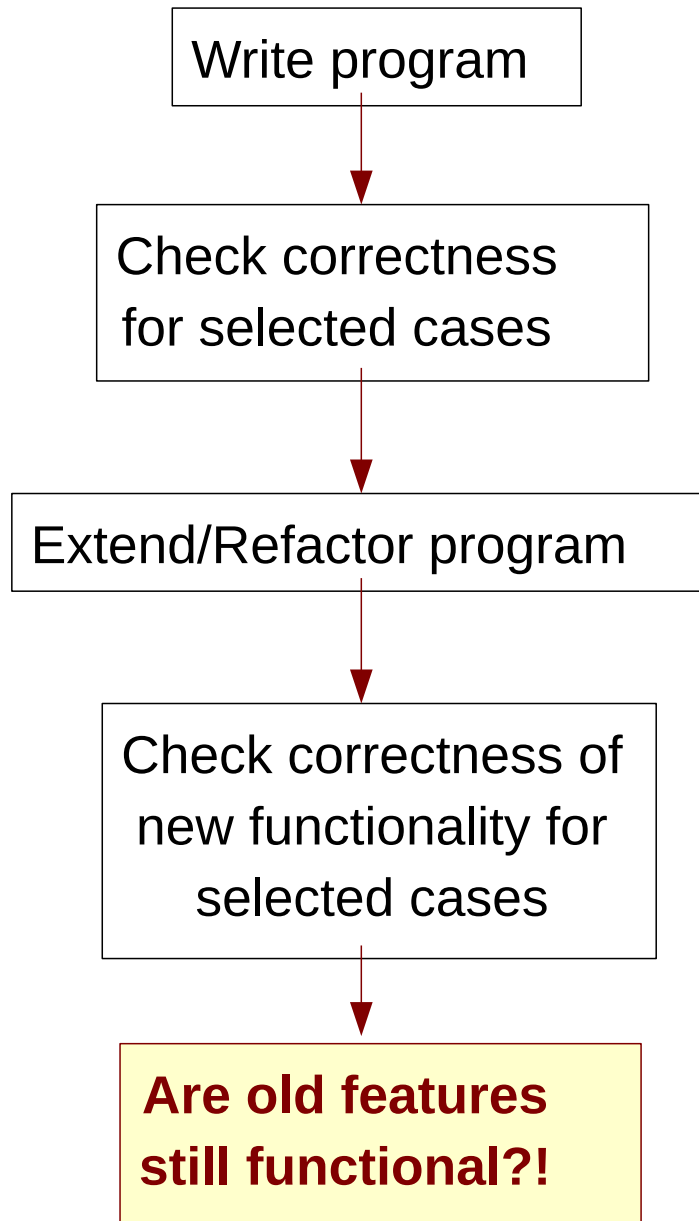
<https://atticlectures.net/scipro/python-2026/>

# Prerequisites

- pytest
- coverage (with pytest binding)

```
conda install pytest coverage pytest-cov
```

# 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

# 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)
- 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

[\[Pytest documentation\]](#)

# Writing simple tests in pytest

```
"""Demo mathematical routines""" mymath.py  
  
def factorial(nn: int) -> int:  
    """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**

```
"""Testing routines for mymath module"""
test_mymath.py

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
- When pytest is imported in an executable, 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
```

```
python -m pytest
```

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

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

# Running tests from VSCode

test\_mymath.py - pytest - Visual Studio Code

```
>test
```

- Cloud Changes: Resume Latest Changes from Cloud
- Python: Configure Tests** (highlighted with a red circle)
- 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
- Terminal: Scroll To Next Command
- Terminal: Scroll To Previous Command
- Terminal: Scroll to Top
- Terminal: Select To Next Command
- Terminal: Select To Next Line
- Terminal: Select To Previous Command

```
1 import mymath
2
3
4 def test_factorial_5():
5     """Test whether 5! is correct"""
6     res = mymath.factorial(5)
7     assert res == 120
8
9
10 def test_factorial_0():
11     """Test whether 0! is correct"""
12     res = mymath.factorial(0)
13     assert res == 1
14
```

test\_mymath.py - pytest - Visual Studio Code

```
Select a test framework/tool to enable
```

- unittest Standard Python test framework  
<https://docs.python.org/3/library/unittest.html>
- pytest pytest framework** (highlighted with a red circle)  
<http://docs.pytest.org/>

```
1 import mymath
2
3
4 def test_factorial_5():
5     """Test whether 5! is correct"""
6     res = mymath.factorial(5)
7     assert res == 120
8
9
10 def test_factorial_0():
11     """Test whether 0! is correct"""
12     res = mymath.factorial(0)
13     assert res == 1
14
```

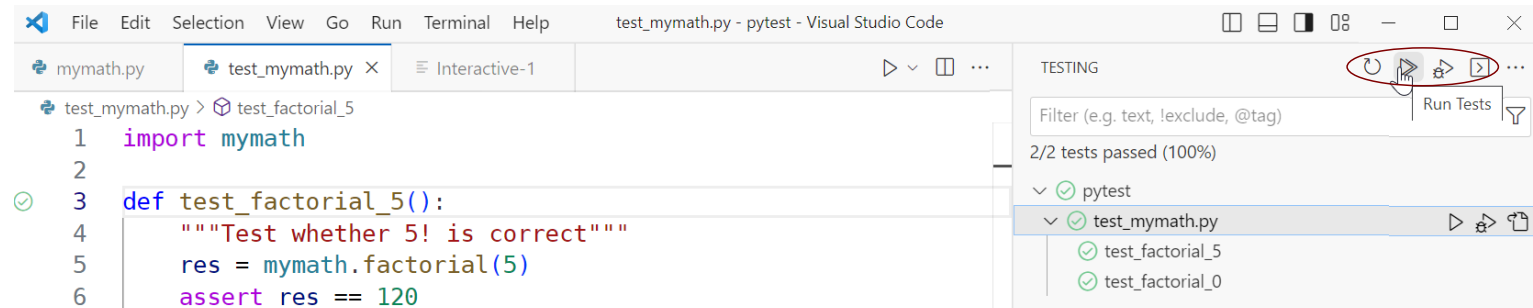
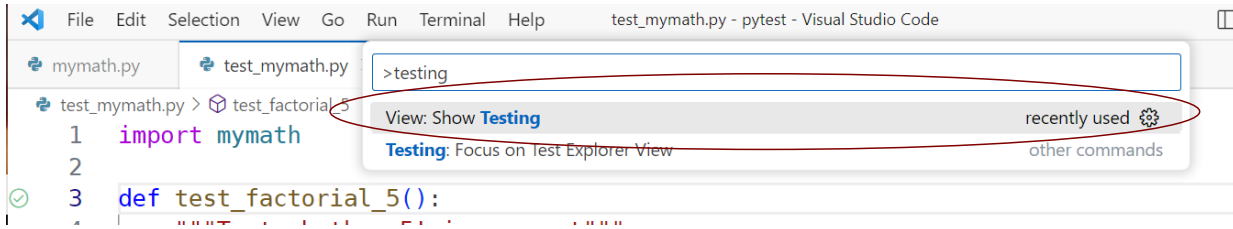
test\_mymath.py - pytest - Visual Studio Code

```
Select the directory containing the tests
```

- . Root directory** (highlighted with a red circle)
- \_\_pycache\_\_

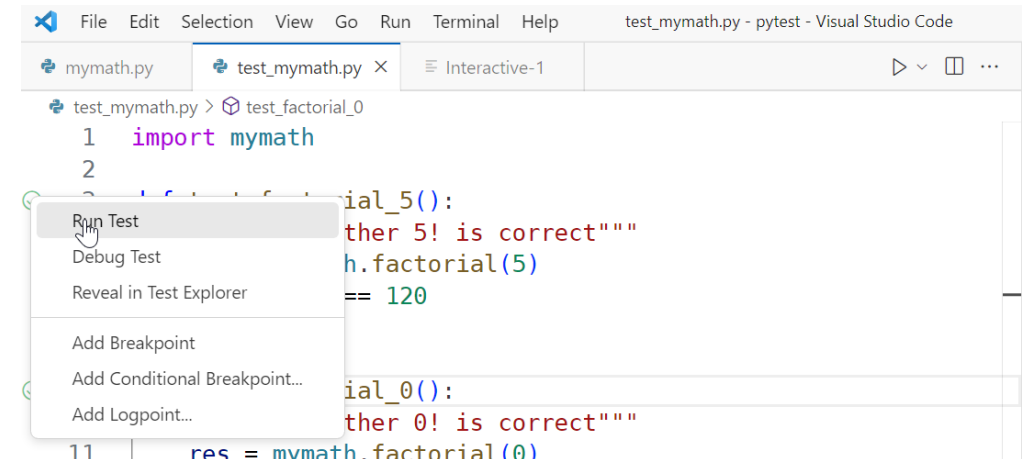
```
1 import mymath
2
3
4 def test_factorial_5():
5     """Test whether 5! is correct"""
6     res = mymath.factorial(5)
7     assert res == 120
8
9
10 def test_factorial_0():
11     """Test whether 0! is correct"""
12     res = mymath.factorial(0)
13     assert res == 1
14
```

# Running tests from VSCode



Run all tests

Run individual test  
(right click)



# Putting tests in a separate directory

- In most projects tests are separated from actual code
- **Convention:** tests are placed in a **test/** subdirectory
- Pytest can still **discover** them, when invoked **from the root project folder**

```
python3 -m pytest
```

```
python -m pytest
```

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

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

- Some IDEs (e.g. Visual Studio Code) need an **empty package initialization file** (`__init__.py`) in the **test/** folder to discover tests properly

```
.gitignore  
README.txt  
solvers.py  
test/
```

```
__init__.py  
test_solver.py
```

Empty file  
(make sure to add  
to repository)

# 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

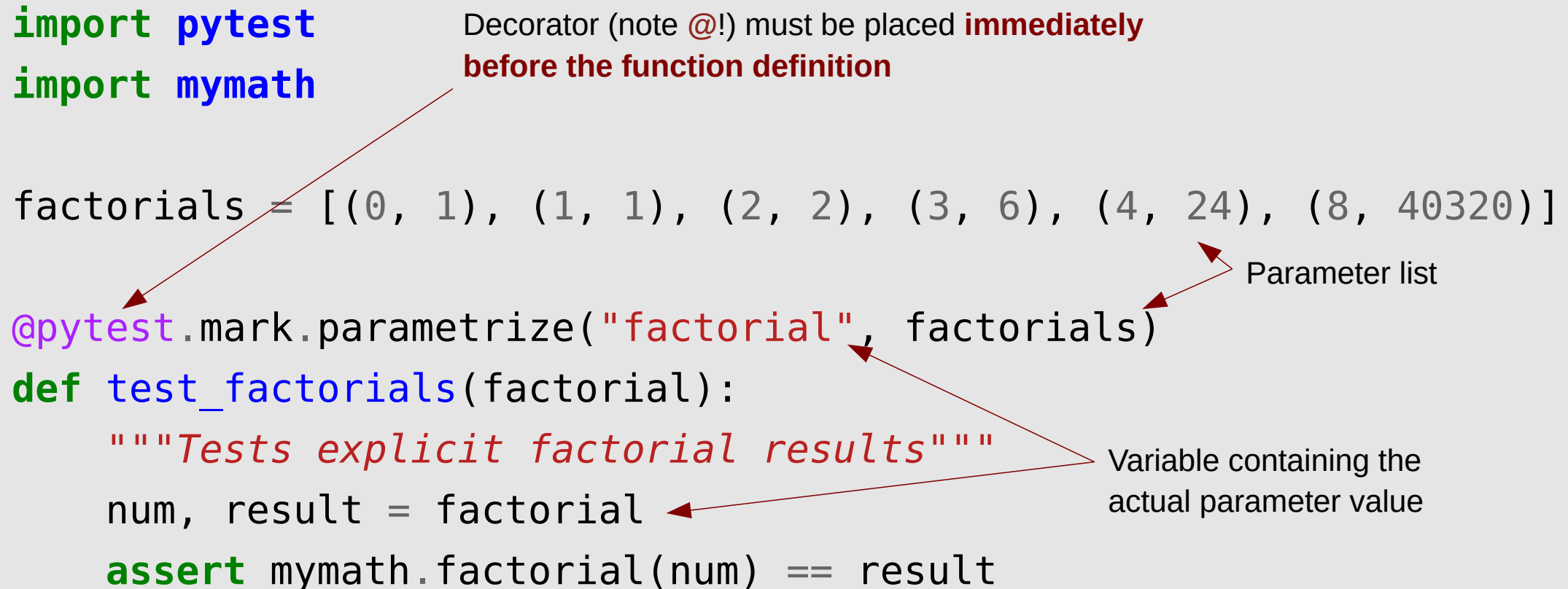
factorials = [(0, 1), (1, 1), (2, 2), (3, 6), (4, 24), (8, 40320)]

@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 files)
- Calculate result using prepared input, compare result with prepared result

```
import pytest
import solvers
TESTNAMES = ['elimination_3', 'pivot_3']

@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.allclose(xx_gauss, xx_expected, rtol=1e-10, atol=1e-10)
```

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

# 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 3<sup>rd</sup> 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
-----				
<code>mymath.py</code>	6	0	100%	
<code>test_mymath.py</code>	27	0	100%	
-----				
TOTAL	33	0	100%	

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

```
coverage html -d coverage_html
```

Directory where  
HTML pages  
should be stored

Coverage report: 98%

<i>Module</i>	<i>statements</i>	<i>missing</i>	<i>excluded</i>	<i>coverage</i>
<u>solvers</u>	25	1	0	96%
test_solvers	25	0	0	100%
<b>Total</b>	<b>50</b>	<b>1</b>	<b>0</b>	<b>98%</b>

Open [coverage\\_html/index.html](coverage_html/index.html) in a browser

*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 ...

# 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 2<sup>nd</sup> or 3<sup>rd</sup> argument** depending on logical values in 1<sup>st</sup>

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

→ [1, 0, 9]

# Comparing floating point values (scalars / arrays)

Floating point values **must not** be compared with the “==” operator

Floating point values **must be compared with tolerances**

## Scalars

```
np.abs(x - y) < ABS_TOL
```

 absolute error

```
np.abs((x - y) / y) < REL_TOL
```

 relative error

```
np.isclose(x, y, rtol=REL_TOL, atol=ABS_TOL)
```

“combined” error

- Floating point representation is **not exact** (e.g. 0.1 requires infinite binary digits)
- Numerical errors** occur during operations

```
6 * 0.1 == 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1
```

 → False

## Arrays

```
np.all(np.abs(x - y) < ABS_TOL)
```

```
np.all(np.abs((x - y) / y) < REL_TOL)
```

```
np.allclose(x, y, rtol=REL_TOL, atol=ABS_TOL)
```