Secure Software Design

UNIVERSITÀ DELLA CALABRIA

il Campus per eccellenza

Introduction to Test-Driven Design

Mario Alviano

Introduction

- Test-Driven Design (TDD) is a methodology
  - It helps to create better code
  - It will not solve all your problems
- Not a religion
  - Do not commit blindly to it
  - Understand it, don’t let it dominate you

These slides are based on Chapter 1 of **Clean Architectures in Python** by *Leonardo Giordani*

https://leanpub.com/clean-architectures-in-python

A real-life example

Boss: I just met with the rest of the board. Our clients are not happy, we didn’t fix enough bugs in the last two months.

Programmer: I see. How many bugs did we fix?

Boss: Well, not enough!

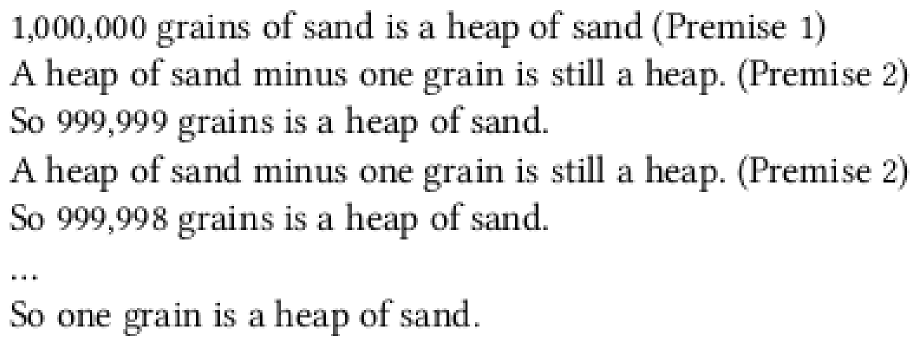
Programmer: OK, so how many bugs do we have to fix every month?

Boss: More!

How to understand if we improved “enough”?

What are we going to measure?!?

Foggy concepts



1,000,000 grains of sand is a heap of sand (Premise 1)
A heap of sand minus one grain is still a heap. (Premise 2)
So 999,999 grains is a heap of sand.
A heap of sand minus one grain is still a heap. (Premise 2)
So 999,998 grains is a heap of sand.
...
So one grain is a heap of sand.

- Avoid to work with nebulous concepts
- Heap here is foggy
- We want to work with very precise concepts
- We must measure something to understand if we improved something!

The idea of TDD

- Write a function and expect it “to work” (?)
  - How do you test if the function “works”?
  - What do you mean by “works”?
- TDD forces you to clearly state your goal **before** you write the code

TDD mantra

Test first, code later

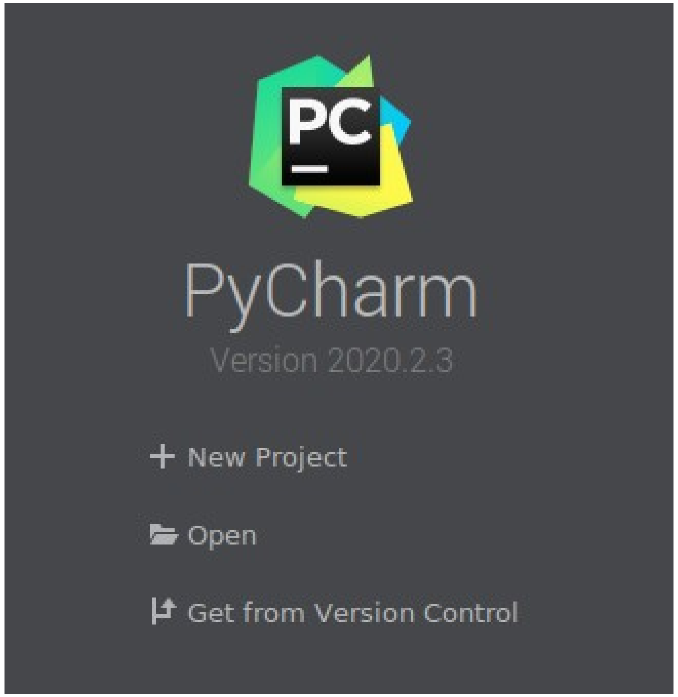
Not just for software, apply it everywhere!

Whatever you are going to do,
you want first to **clearly define your goals** and a reproducible procedure to measure your achievements

Example of test

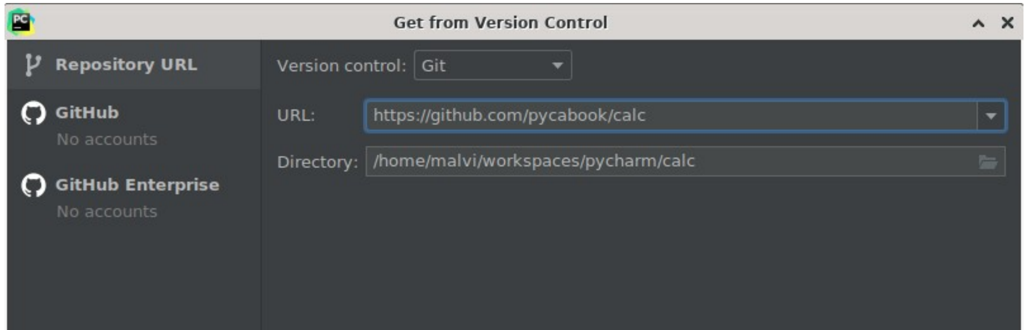
- sum(4, 5) == 9
  - There will be a sum function available in the system
  - The function accepts two integers
  - If the two integers are 4 and 5, the function returns 9
- But if “test first, code later”, then the test will fail
  - True, and expected
  - The test is an evidence that some feature is missing in the system

A simple TDD project



- Let’s apply TDD to create a calculator
- First, let’s check the final result
- Create a new project with PyCharm from the following git repository:

https://github.com/pycabook/calc

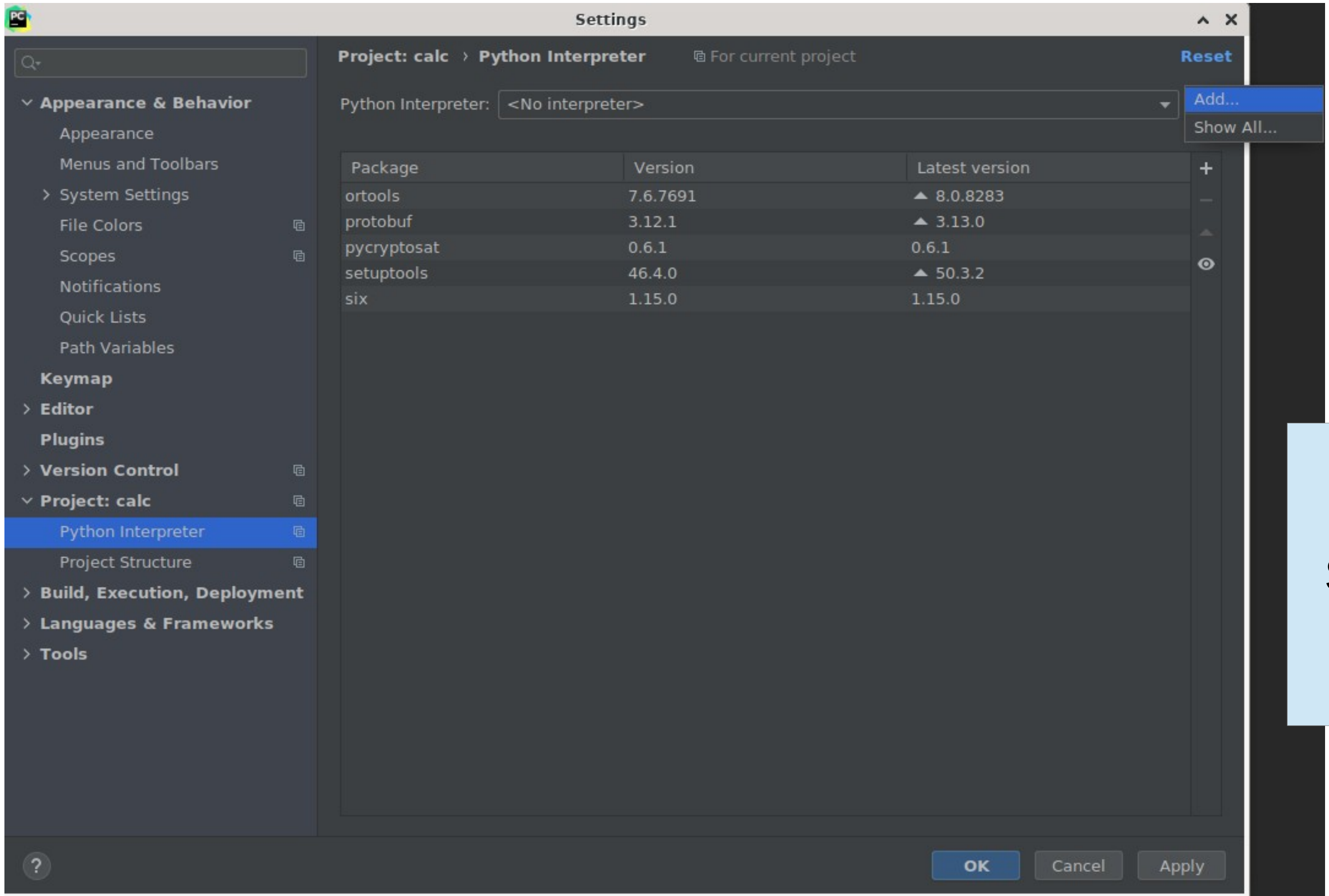


Version control: Git

URL: https://github.com/pycabook/calc

Directory: /home/malvi/workspaces/pycharm/calc

Use a virtual environment



Settings

Project: calc › Python Interpreter   For current project   Reset

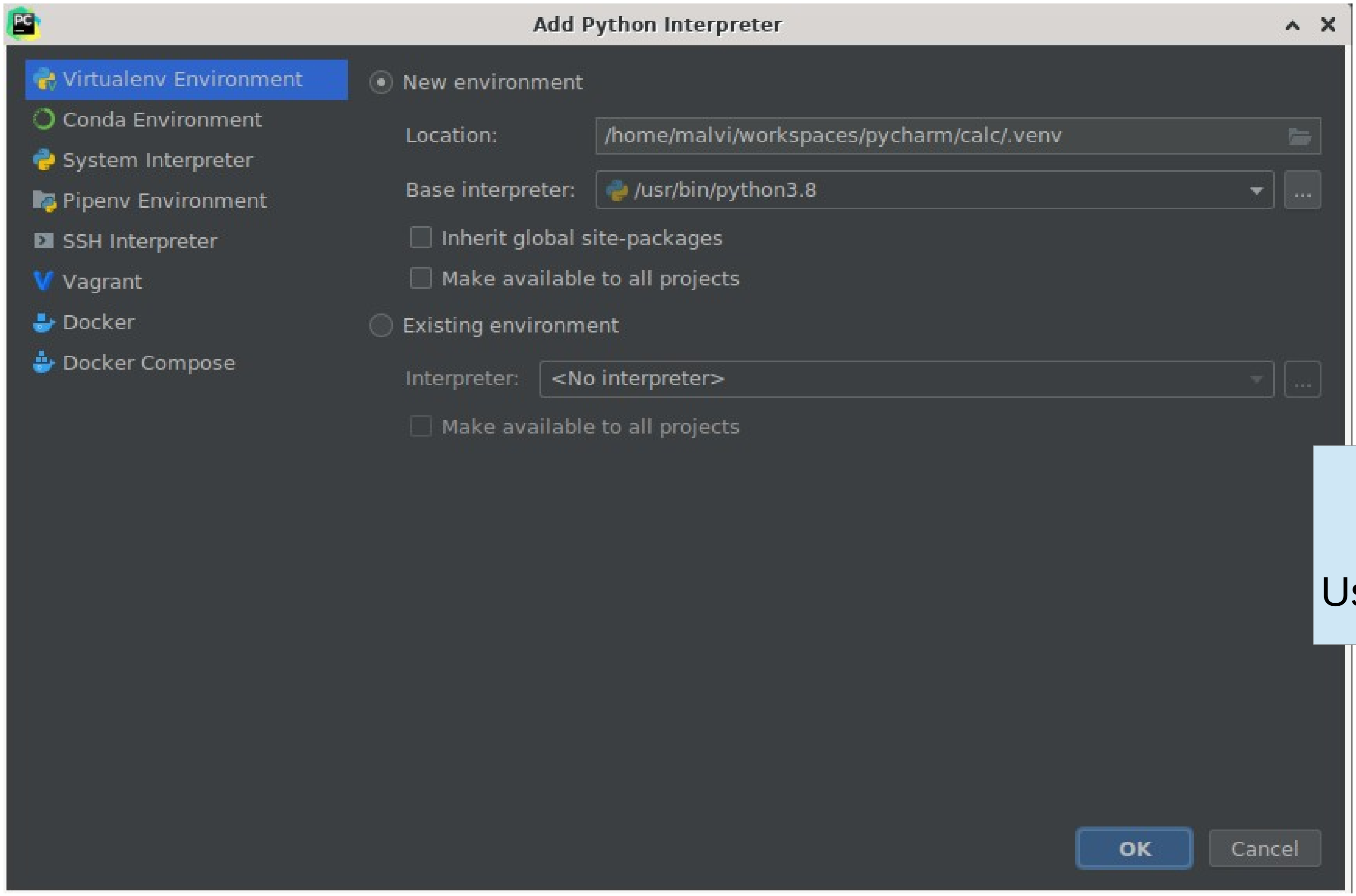
Python Interpreter: <No interpreter>

Package	Version	Latest version
ortools	7.6.7691	▲ 8.0.8283
protobuf	3.12.1	▲ 3.13.0
pycryptosat	0.6.1	0.6.1
setuptools	46.4.0	▲ 50.3.2
six	1.15.0	1.15.0

Menu **File | Settings**

Select Project Python Interpreter

Add a new Python Interpreter


Add Python Interpreter

- Virtualenv Environment
- Conda Environment
- System Interpreter
- Pipenv Environment
- SSH Interpreter
- Vagrant
- Docker
- Docker Compose

(●) New environment

Location: /home/malvi/workspaces/pycharm/calc/.venv

Base interpreter: /usr/bin/python3.8

☐ Inherit global site-packages

☐ Make available to all projects

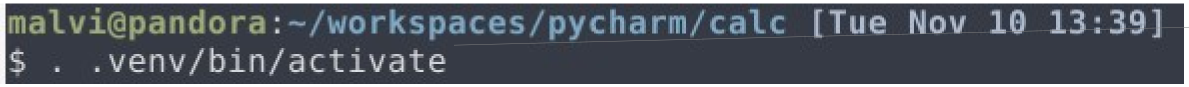
( ) Existing environment

Interpreter: <No interpreter>

☐ Make available to all projects

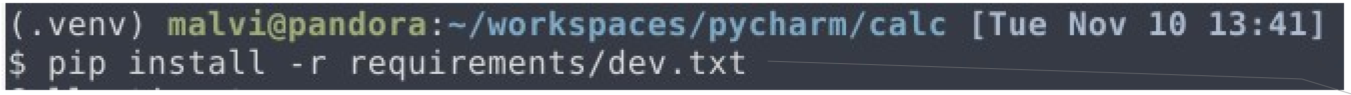
Let’s use pip

Usually .venv directory



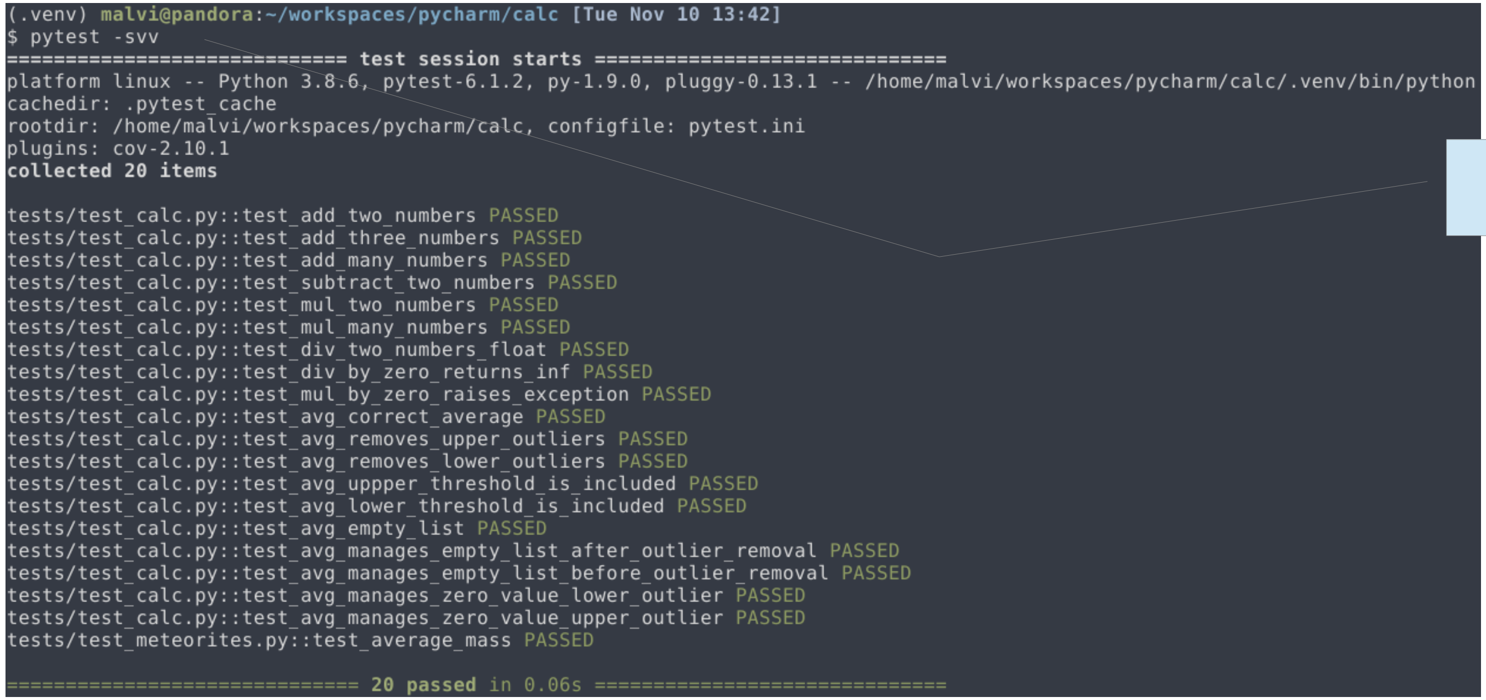
```
malvi@pandora:~/workspaces/pycharm/calc [Tue Nov 10 13:39]
$ . .venv/bin/activate
```

Activate the environment



```
(.venv) malvi@pandora:~/workspaces/pycharm/calc [Tue Nov 10 13:41]
$ pip install -r requirements/dev.txt
```

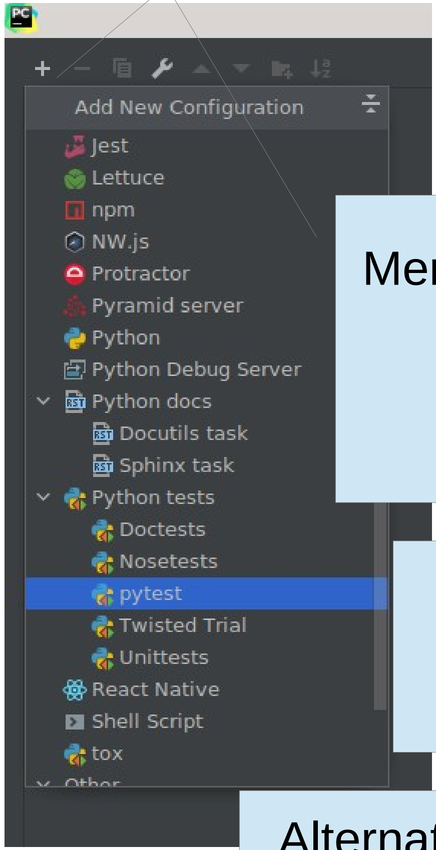
Install all requirements

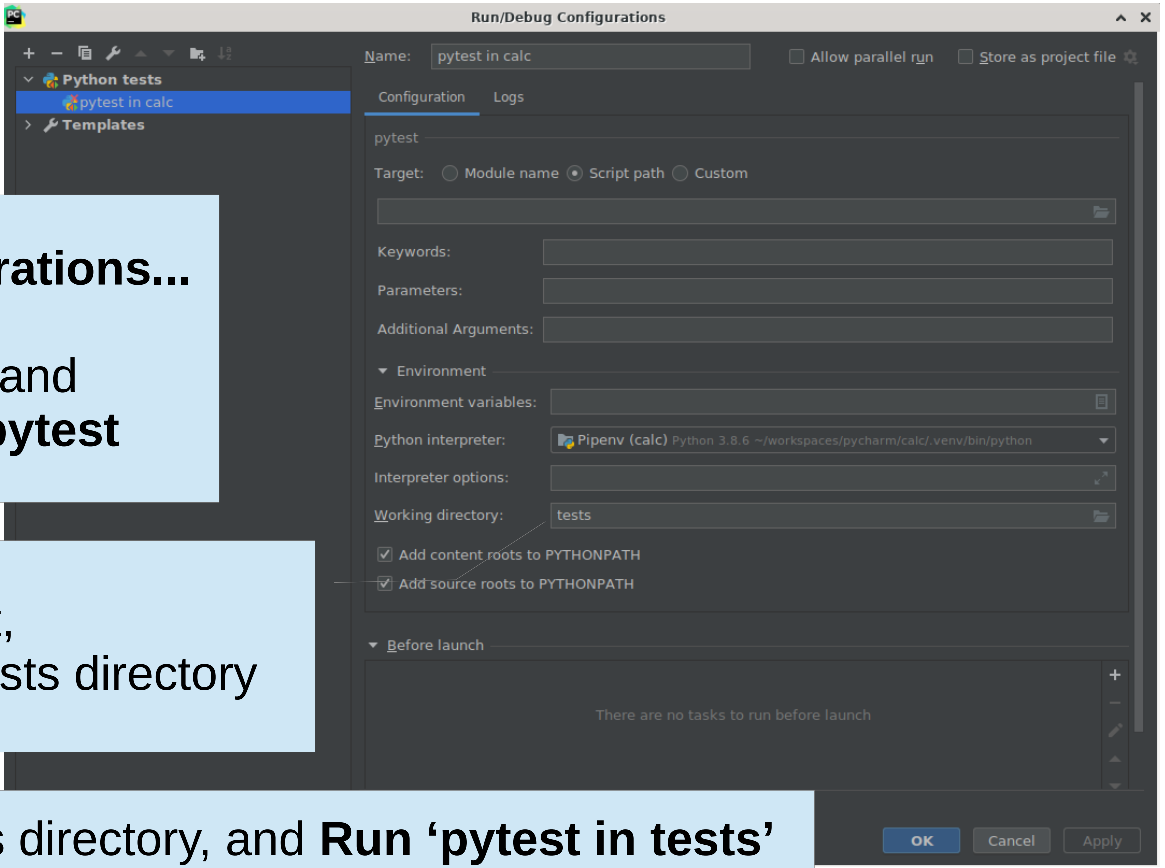


```
(.venv) malvi@pandora:~/workspaces/pycharm/calc [Tue Nov 10 13:42]
$ pytest -svv
============================= test session starts ==============================
platform linux -- Python 3.8.6, pytest-6.1.2, py-1.9.0, pluggy-0.13.1 -- /home/malvi/workspaces/pycharm/calc/.venv/bin/python
cachedir: .pytest_cache
rootdir: /home/malvi/workspaces/pycharm/calc, configfile: pytest.ini
plugins: cov-2.10.1
collected 20 items

tests/test_calc.py::test_add_two_numbers PASSED
tests/test_calc.py::test_add_three_numbers PASSED
tests/test_calc.py::test_add_many_numbers PASSED
tests/test_calc.py::test_subtract_two_numbers PASSED
tests/test_calc.py::test_mul_two_numbers PASSED
tests/test_calc.py::test_mul_many_numbers PASSED
tests/test_calc.py::test_div_two_numbers_float PASSED
tests/test_calc.py::test_div_by_zero_returns_inf PASSED
tests/test_calc.py::test_mul_by_zero_raises_exception PASSED
tests/test_calc.py::test_avg_correct_average PASSED
tests/test_calc.py::test_avg_removes_upper_outliers PASSED
tests/test_calc.py::test_avg_removes_lower_outliers PASSED
tests/test_calc.py::test_avg_uppper_threshold_is_included PASSED
tests/test_calc.py::test_avg_lower_threshold_is_included PASSED
tests/test_calc.py::test_avg_empty_list PASSED
tests/test_calc.py::test_avg_manages_empty_list_after_outlier_removal PASSED
tests/test_calc.py::test_avg_manages_empty_list_before_outlier_removal PASSED
tests/test_calc.py::test_avg_manages_zero_value_lower_outlier PASSED
tests/test_calc.py::test_avg_manages_zero_value_upper_outlier PASSED
tests/test_meteorites.py::test_average_mass PASSED

============================== 20 passed in 0.06s ==============================
```

Run all tests

Run tests with PyCharm





Name: pytest in calc
☐ Allow parallel run
☐ Store as project file

pytest

Target: ○ Module name ● Script path ○ Custom

Keywords:

Parameters:

Additional Arguments:

Environment

Environment variables:

Python interpreter: Pipenv (calc) Python 3.8.6 ~/workspaces/pycharm/calc/.venv/bin/python

Interpreter options:

Working directory: tests

☑ Add content roots to PYTHONPATH

☑ Add source roots to PYTHONPATH

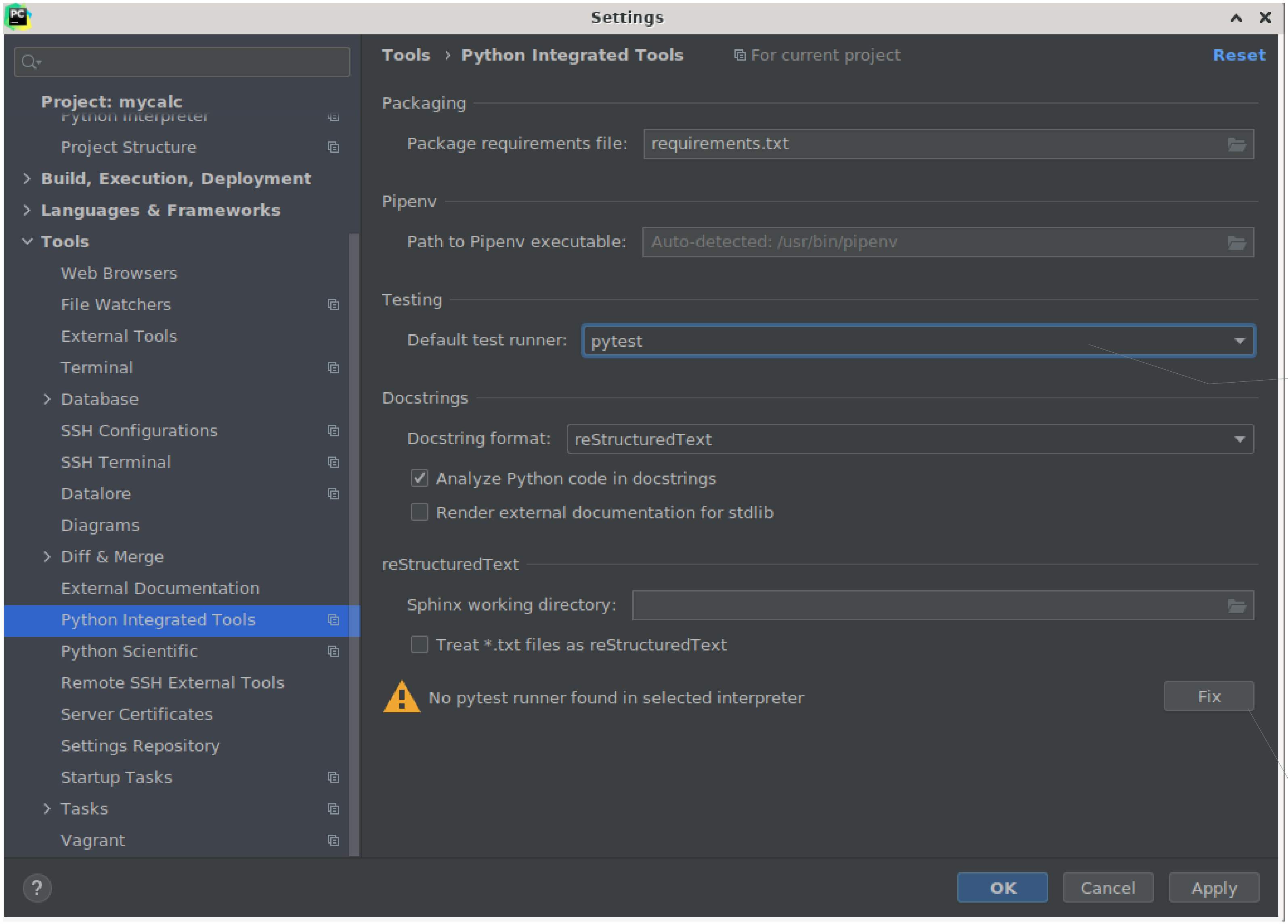
Before launch

Menu **Run | Edit Configurations...**

Click the plus button and select **Python tests | pytest**

If you want, restrict pytest to the tests directory

Alternative: right-click the tests directory, and **Run ‘pytest in tests’**



Packaging

Package requirements file: requirements.txt

Pipenv

Path to Pipenv executable: Auto-detected: /usr/bin/pipenv

Testing

Default test runner: pytest

Docstrings

Docstring format: reStructuredText

☑ Analyze Python code in docstrings

☐ Render external documentation for stdlib

reStructuredText

Sphinx working directory:

☐ Treat *.txt files as reStructuredText

No pytest runner found in selected interpreter — Fix

If you don't see pytest, menu File | Settings, Tools | Python Integrated Tools

Select pytest here

Click here

Requirements

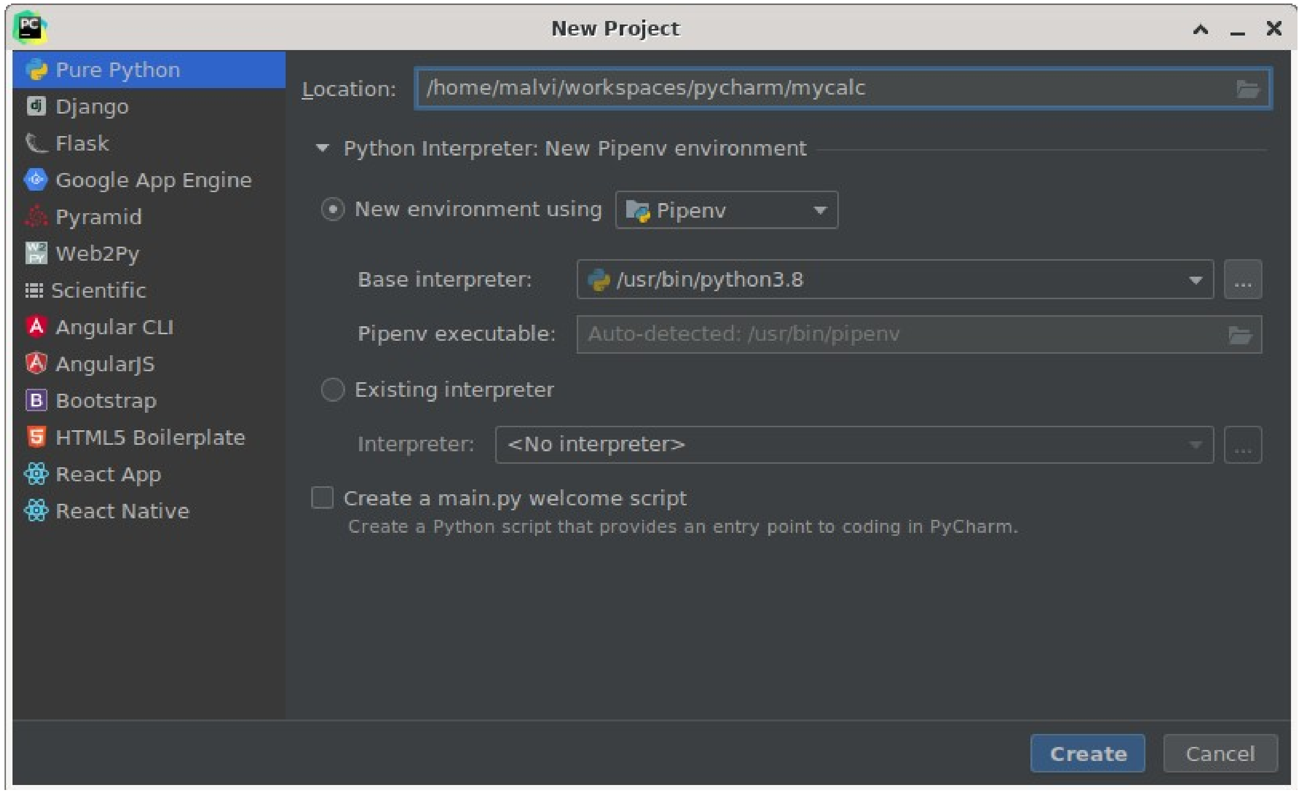
The goal of the project is to write a class Calc that performs calculations: addition, subtraction, multiplication, and division

- Addition and multiplication shall accept multiple arguments
- Division shall return a float value, and division by zero shall return the string "inf"
- Multiplication by zero must raise a ValueError exception

The class also provides a function to compute the average of an iterable like a list

- This function gets two optional upper and lower thresholds and should remove from the computation the values that fall outside these boundaries
- For an empty sequence, the average is undefined and the function returns None

Requirements on multiplication and division are strange… it’s just an example!



Location: /home/malvi/workspaces/pycharm/mycalc

Python Interpreter: New Pipenv environment

[✓ New environment using] Pipenv

Base interpreter: /usr/bin/python3.8

Pipenv executable: Auto-detected: /usr/bin/pipenv

Existing interpreter

Interpreter: <No interpreter>

Create a main.py welcome script

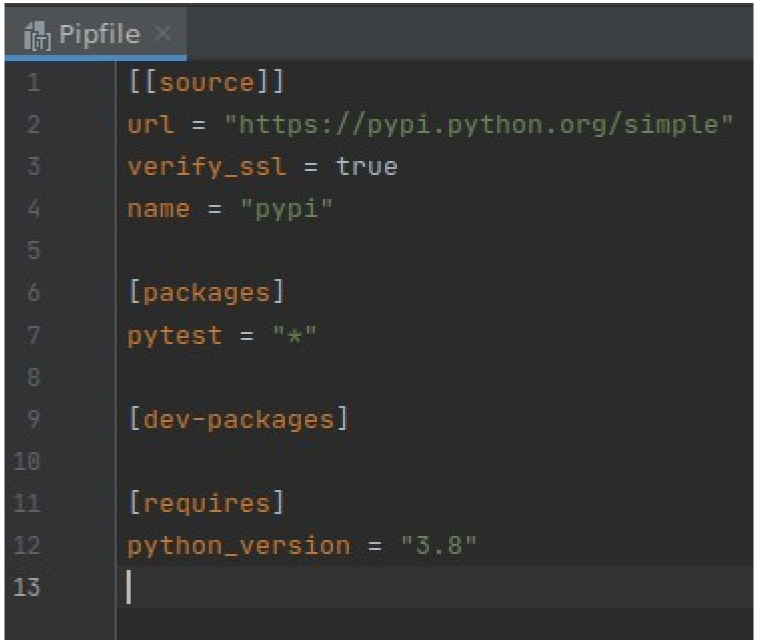
Create a Python script that provides an entry point to coding in PyCharm.

Start a new project

Use pip

Set pytest for tests

Add a tests directory



```
[[source]]
url = "https://pypi.python.org/simple"
verify_ssl = true
name = "pypi"

[packages]
pytest = "*"

[dev-packages]

[requires]
python_version = "3.8"

```

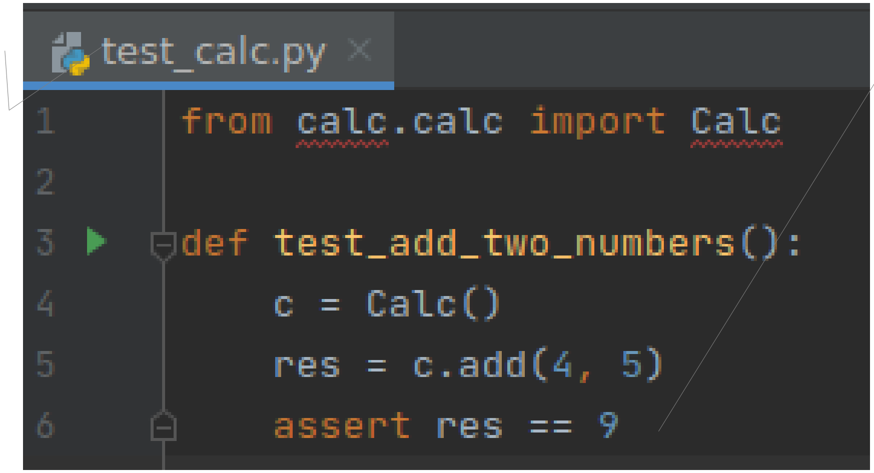
Pipfile should look like this one

Your first test

- Create file tests/test_calc.py with the following content:

pytest discovers tests

Every test_* function is a test

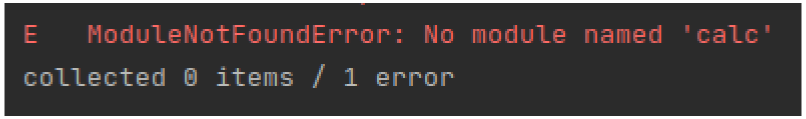


test_calc.py

```python
from calc.calc import Calc

def test_add_two_numbers():
    c = Calc()
    res = c.add(4, 5)
    assert res == 9
```

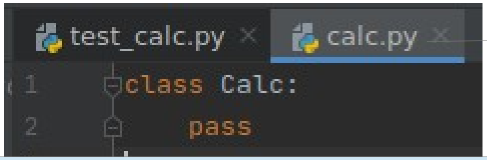
A test fails if it raises an exception

Have you noted the errors? They are expected! We are going to fix them now



```
E    ModuleNotFoundError: No module named 'calc'
collected 0 items / 1 error
```

Let’s fix the failing test

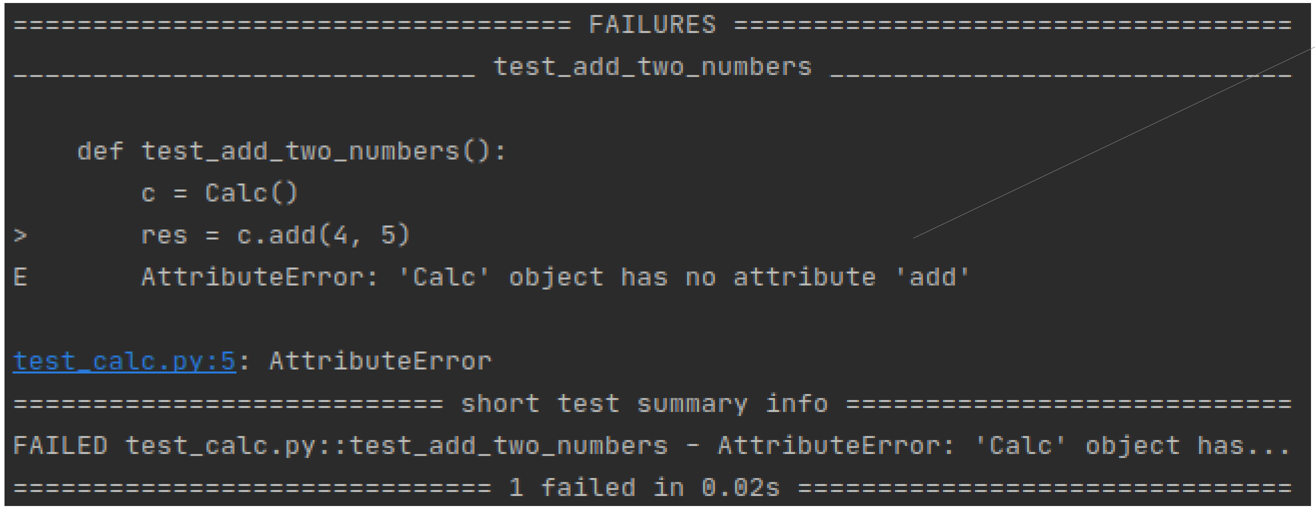


```
class Calc:
    pass
```

Create file calc/calc.py

I know, the class is very minimal, it’s not implementing the requirements

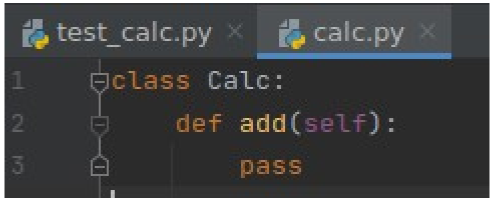
TDD: The requirements are used to write the tests, the tests are used to write the code.



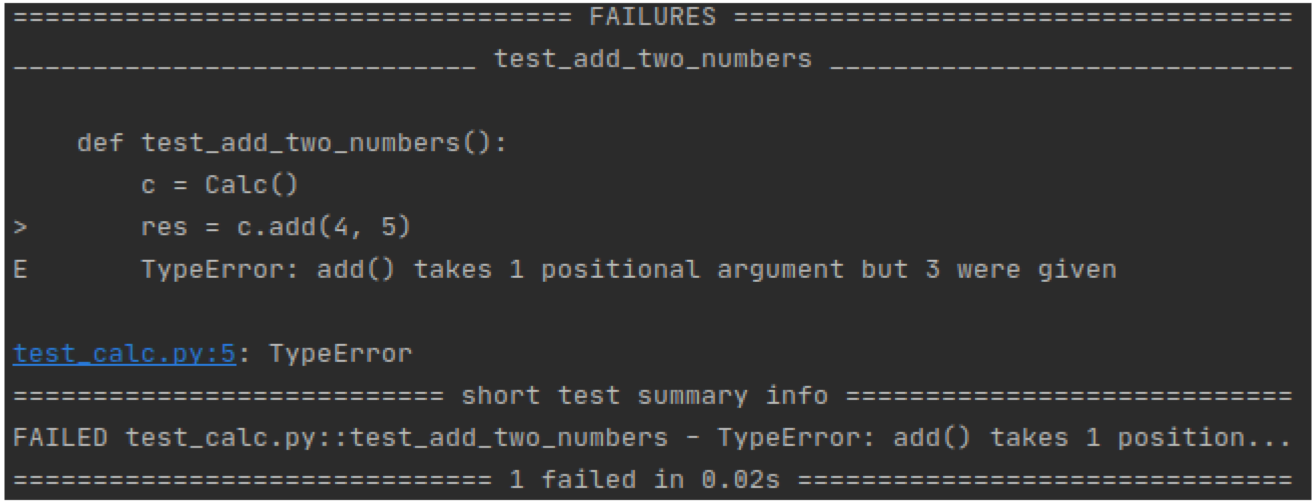
```
=================================== FAILURES ====================================
_____________________________ test_add_two_numbers _______________________________

    def test_add_two_numbers():
        c = Calc()
>       res = c.add(4, 5)
E       AttributeError: 'Calc' object has no attribute 'add'

test_calc.py:5: AttributeError
=========================== short test summary info ============================
FAILED test_calc.py::test_add_two_numbers - AttributeError: 'Calc' object has...
============================== 1 failed in 0.02s ===============================
```

Our first test still fails

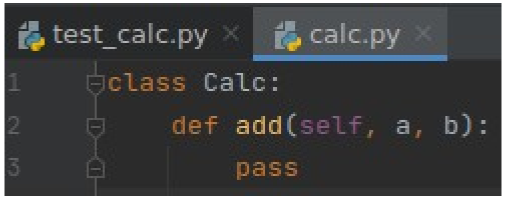
Class Calc has no add method


test_calc.py calc.py

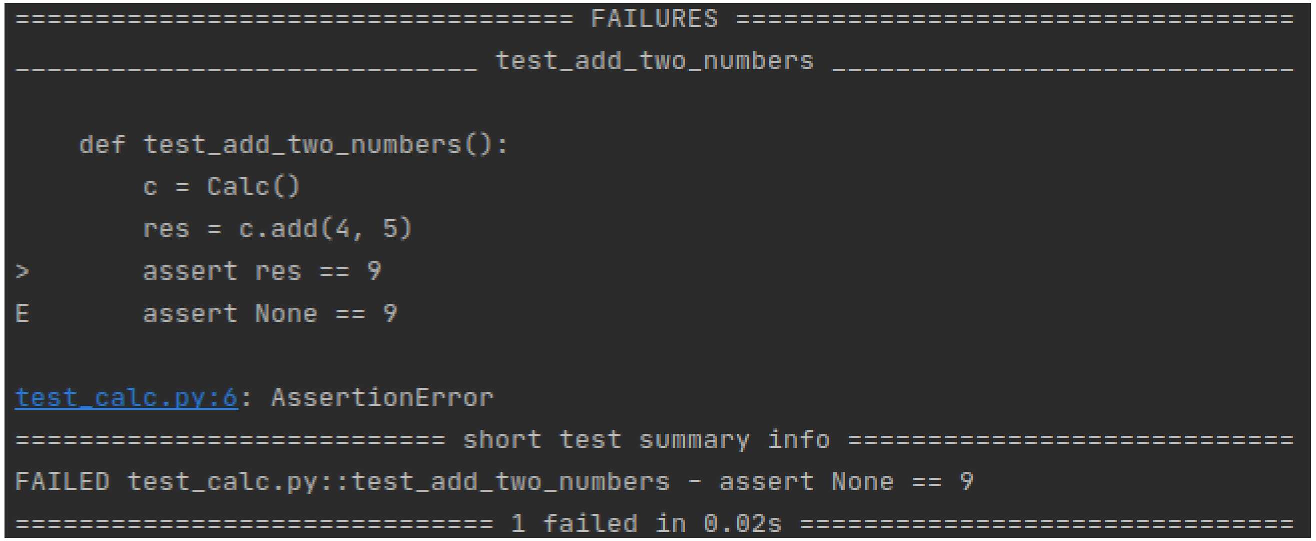
```
class Calc:
    def add(self):
        pass
```



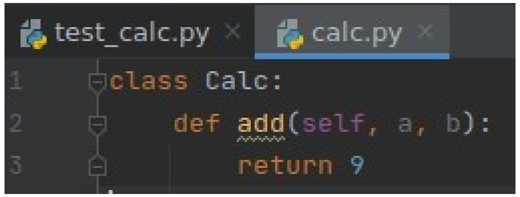
```
==================================== FAILURES ====================================
______________________________ test_add_two_numbers ______________________________

    def test_add_two_numbers():
        c = Calc()
>       res = c.add(4, 5)
E       TypeError: add() takes 1 positional argument but 3 were given

test_calc.py:5: TypeError
=========================== short test summary info ============================
FAILED test_calc.py::test_add_two_numbers - TypeError: add() takes 1 position...
============================== 1 failed in 0.02s ===============================
```


test_calc.py calc.py

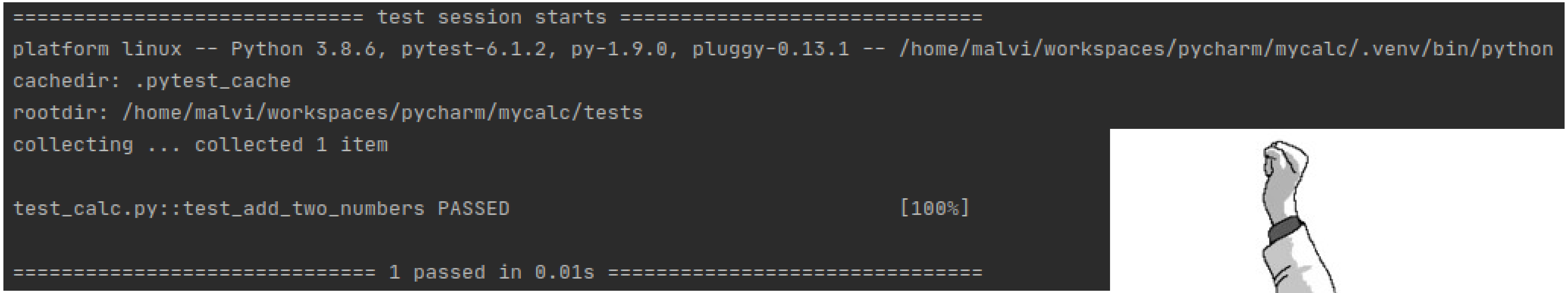
```
class Calc:
    def add(self, a, b):
        pass
```



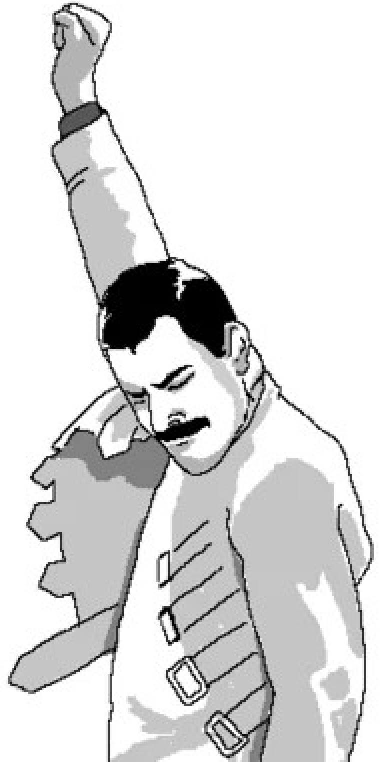
```
==================================== FAILURES ====================================
______________________________ test_add_two_numbers ______________________________

    def test_add_two_numbers():
        c = Calc()
        res = c.add(4, 5)
>       assert res == 9
E       assert None == 9

test_calc.py:6: AssertionError
=========================== short test summary info ============================
FAILED test_calc.py::test_add_two_numbers - assert None == 9
============================== 1 failed in 0.02s ===============================
```



```python
class Calc:
    def add(self, a, b):
        return 9
```



```
============================= test session starts ==============================
platform linux -- Python 3.8.6, pytest-6.1.2, py-1.9.0, pluggy-0.13.1 -- /home/malvi/workspaces/pycharm/mycalc/.venv/bin/python
cachedir: .pytest_cache
rootdir: /home/malvi/workspaces/pycharm/mycalc/tests
collecting ... collected 1 item

test_calc.py::test_add_two_numbers PASSED                                [100%]

============================== 1 passed in 0.01s ===============================
```



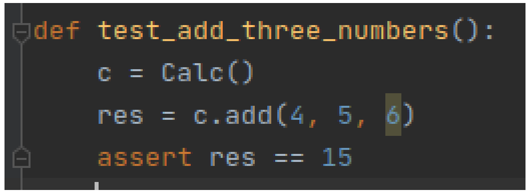
Stop here! All tests are satisfied

Do you want more? Give me more tests!

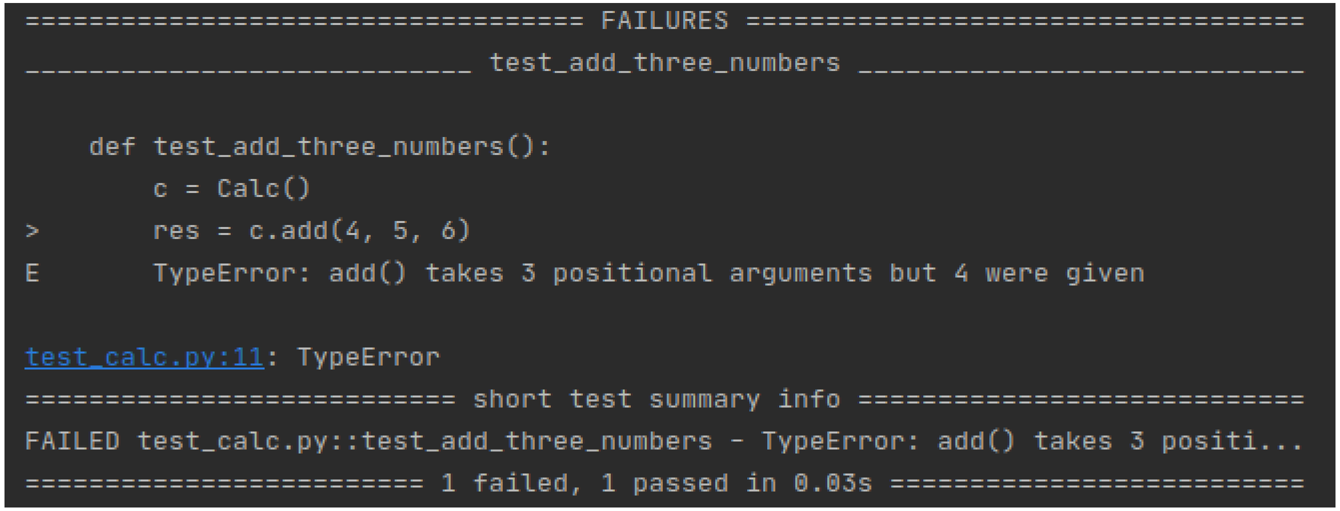
Obviously, here we are exaggerating,
but it’s just to give the idea

Who said more tests?

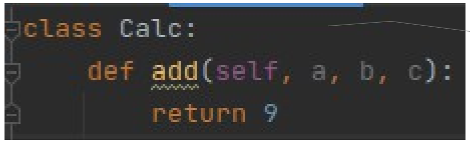
- Requirement: Addition and multiplication shall accept multiple arguments
  - Not only two, but possibly three, four, and so on
- Let’s add a test to tests/test_calc.py



```
def test_add_three_numbers():
    c = Calc()
    res = c.add(4, 5, 6)
    assert res == 15
```

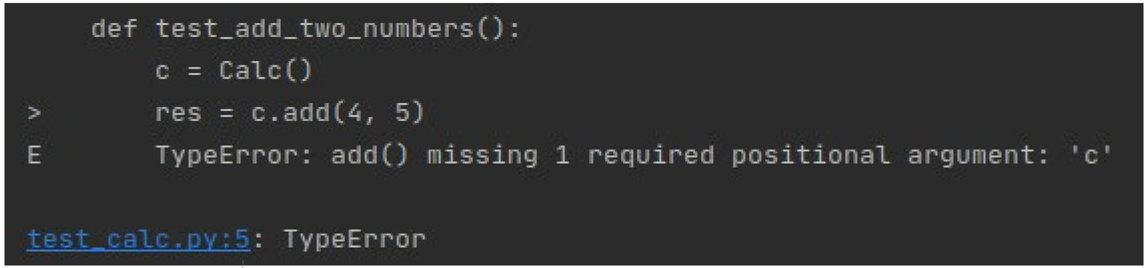


```
=================================== FAILURES ===================================
____________________________ test_add_three_numbers ____________________________

    def test_add_three_numbers():
        c = Calc()
>       res = c.add(4, 5, 6)
E       TypeError: add() takes 3 positional arguments but 4 were given

test_calc.py:11: TypeError
=========================== short test summary info ============================
FAILED test_calc.py::test_add_three_numbers - TypeError: add() takes 3 positi...
========================= 1 failed, 1 passed in 0.03s ==========================
```



```
class Calc:
    def add(self, a, b, c):
        return 9
```

Minimum code fix the failing test: add a third argument



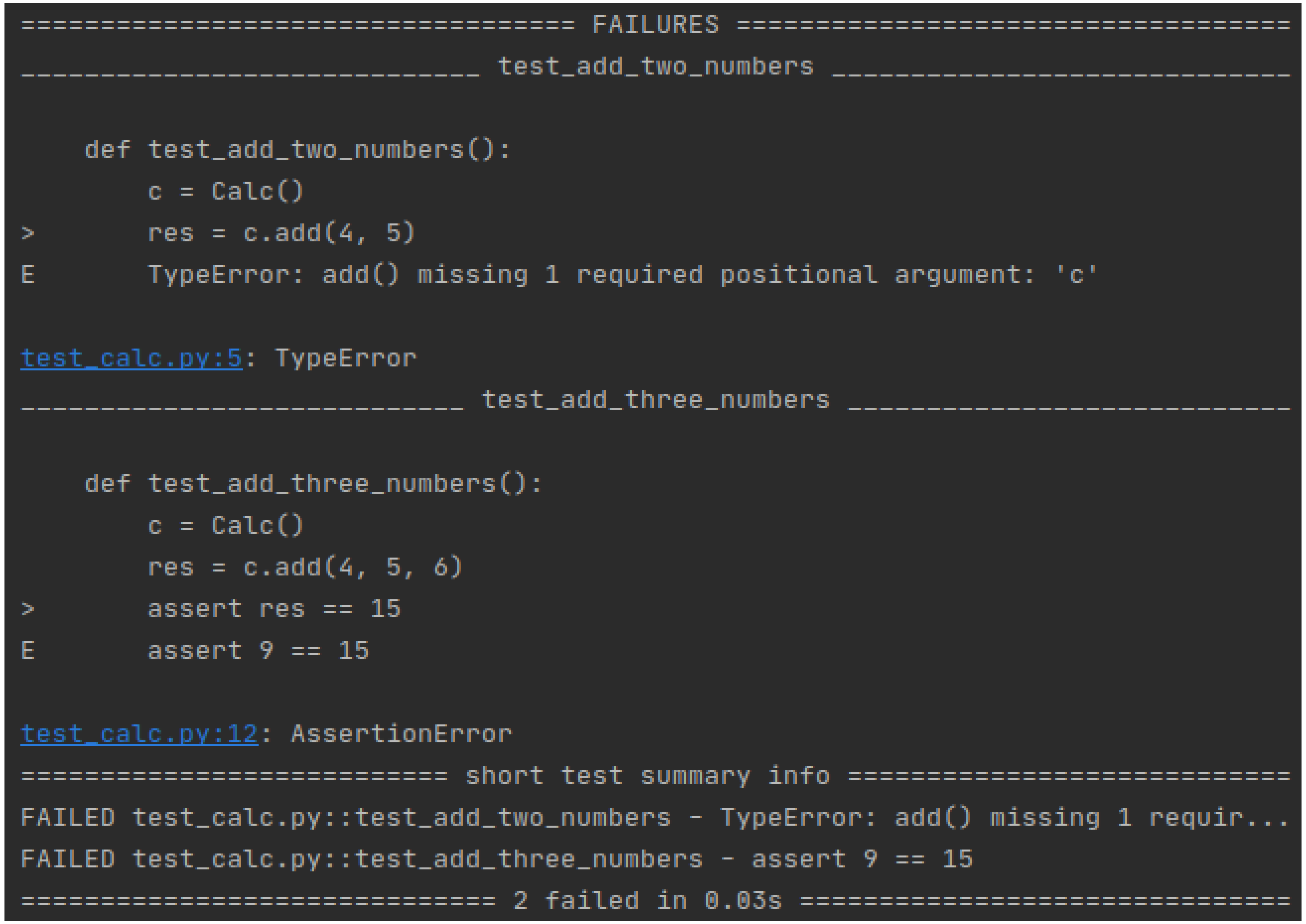
```
    def test_add_two_numbers():
        c = Calc()
>       res = c.add(4, 5)
E       TypeError: add() missing 1 required positional argument: 'c'

test_calc.py:5: TypeError
```

Oops! We just broke test 1

That’s good! We call it a regression test fail

We know that the problem was introduced in the last change

What if test 1 was not here to help?

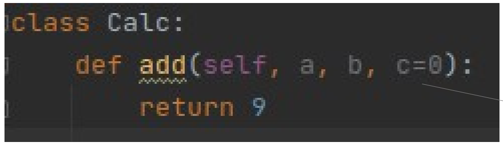


```
=================================== FAILURES ===================================
____________________________ test_add_two_numbers _____________________________

    def test_add_two_numbers():
        c = Calc()
>       res = c.add(4, 5)
E       TypeError: add() missing 1 required positional argument: 'c'

test_calc.py:5: TypeError
___________________________ test_add_three_numbers ____________________________

    def test_add_three_numbers():
        c = Calc()
        res = c.add(4, 5, 6)
>       assert res == 15
E       assert 9 == 15

test_calc.py:12: AssertionError
=========================== short test summary info ============================
FAILED test_calc.py::test_add_two_numbers - TypeError: add() missing 1 requir...
FAILED test_calc.py::test_add_three_numbers - assert 9 == 15
============================== 2 failed in 0.03s ===============================
```

Actually, also test 2 is failing

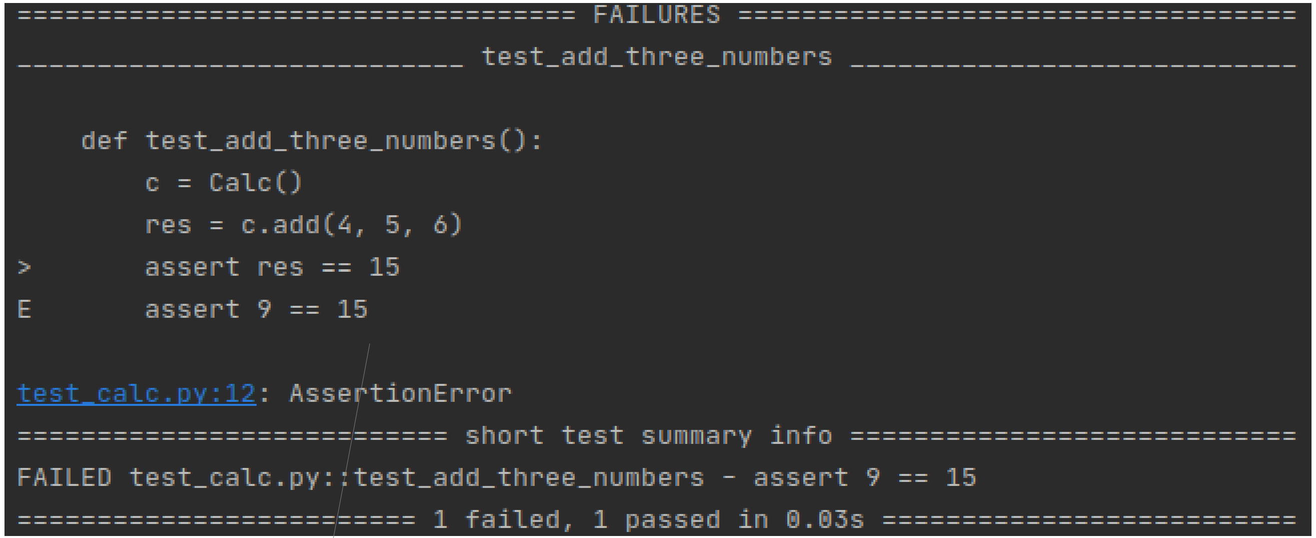
Focus on one failing test
at time

Previously passing tests
should get priority
(they are easier to fix;
just unroll the last change)



```
class Calc:
    def add(self, a, b, c=0):
        return 9
```

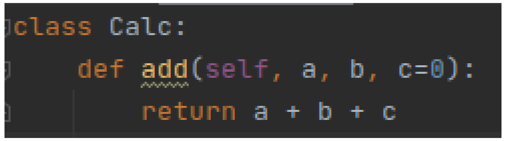
Let’s add a third argument with default value



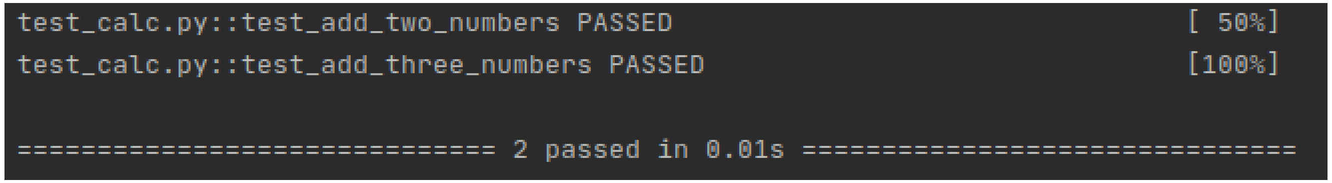
```
==================================== FAILURES ====================================
_____________________________ test_add_three_numbers _____________________________

    def test_add_three_numbers():
        c = Calc()
        res = c.add(4, 5, 6)
>       assert res == 15
E       assert 9 == 15

test_calc.py:12: AssertionError
=========================== short test summary info ============================
FAILED test_calc.py::test_add_three_numbers - assert 9 == 15
========================= 1 failed, 1 passed in 0.03s ==========================
```

Test 1 passes

Test 2 fails: fix with return 15?
Would broke test 1, so no



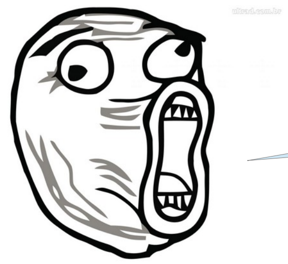
```
class Calc:
    def add(self, a, b, c=0):
        return a + b + c
```



```
test_calc.py::test_add_two_numbers PASSED                                        [ 50%]
test_calc.py::test_add_three_numbers PASSED                                      [100%]

============================== 2 passed in 0.01s ===============================
```

TDD is slow

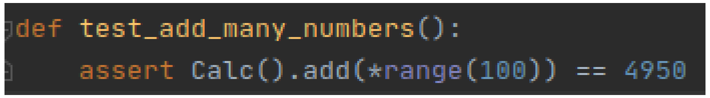
- Yes, it is
- You would do must faster without tests
- Until something will break
  - Time to search for the bug
  - How long was the bug there?
  - How do you find it? You have to write examples



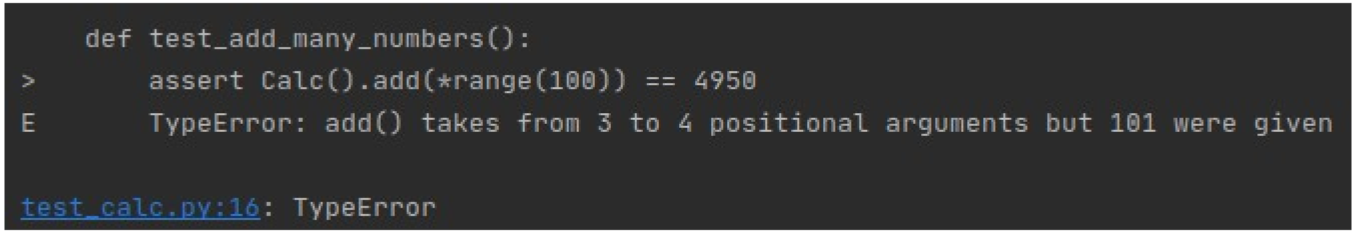
Those examples are tests!!!
Wasn’t better to write them once and for all?

We are not done

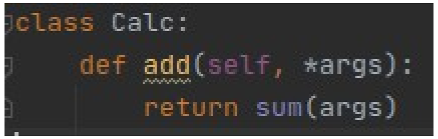
- Requirement: Addition and multiplication shall accept multiple arguments
  - Not only two, but possibly three, four, and so on
- We cannot test infinitely many cases
- We should test at least boundary cases
  - What are the **corner-cases** of our algorithm?
- Example: input from 1 to 100
  - You may not need to test 42
  - But you should test 1 and 100
  - And you should also test for errors, when 0 or 101 are used



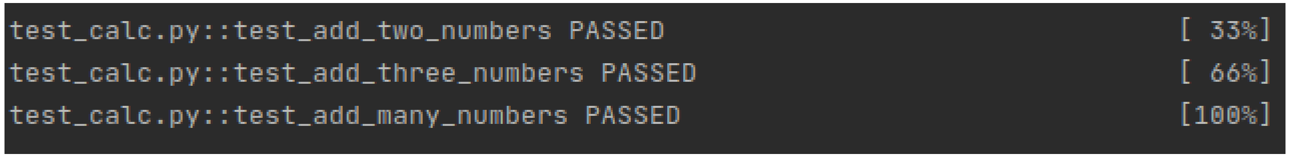
```
def test_add_many_numbers():
    assert Calc().add(*range(100)) == 4950
```



```
    def test_add_many_numbers():
>       assert Calc().add(*range(100)) == 4950
E       TypeError: add() takes from 3 to 4 positional arguments but 101 were given

test_calc.py:16: TypeError
```



```
class Calc:
    def add(self, *args):
        return sum(args)
```



```
test_calc.py::test_add_two_numbers PASSED                                   [ 33%]
test_calc.py::test_add_three_numbers PASSED                                 [ 66%]
test_calc.py::test_add_many_numbers PASSED                                  [100%]
```

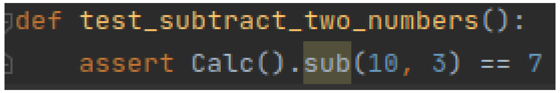
In TDD a solution is not correct when it is beautiful, when it is smart, or when it uses the latest feature of the language

TDD wants your code to pass the tests

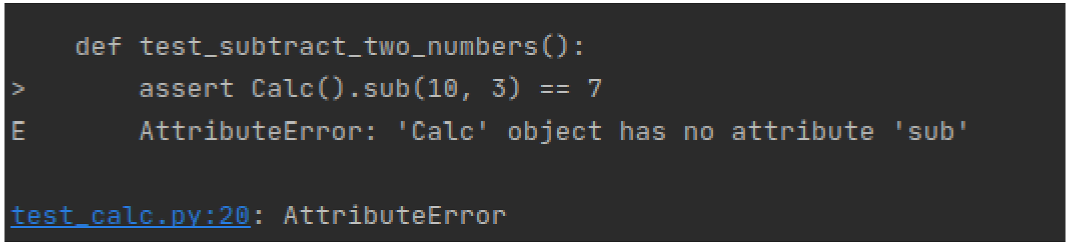
TDD doesn’t cover all the needs of your software project: your code might be ugly, convoluted, and slow

Subtraction

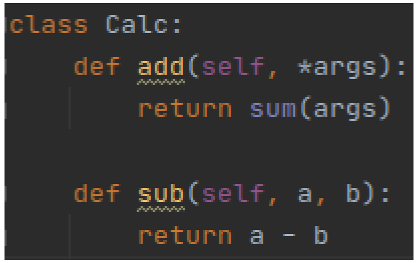
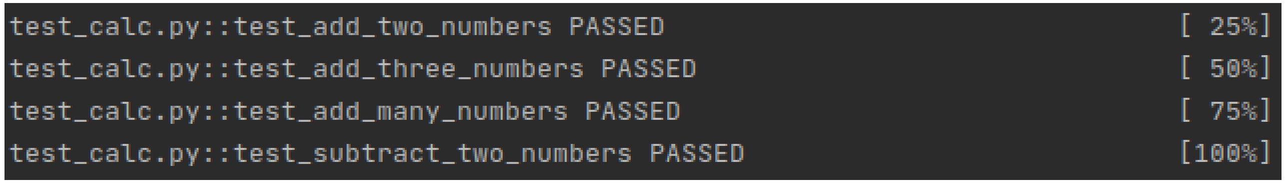
- We need a function to implement subtraction
  - Multiple arguments are not mentioned
  - We limit to two operands
  - We write a test from the requirement



```
def test_subtract_two_numbers():
    assert Calc().sub(10, 3) == 7
```



```
    def test_subtract_two_numbers():
>       assert Calc().sub(10, 3) == 7
E       AttributeError: 'Calc' object has no attribute 'sub'

test_calc.py:20: AttributeError
```


```python
class Calc:
    def add(self, *args):
        return sum(args)

    def sub(self, a, b):
        return a - b
```

```
test_calc.py::test_add_two_numbers PASSED                                  [ 25%]
test_calc.py::test_add_three_numbers PASSED                                [ 50%]
test_calc.py::test_add_many_numbers PASSED                                 [ 75%]
test_calc.py::test_subtract_two_numbers PASSED                             [100%]
```

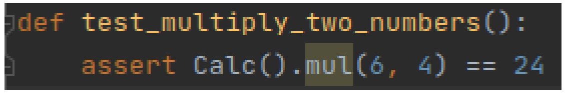
The fix is simple

We don’t really need to do all the passages we did for addition

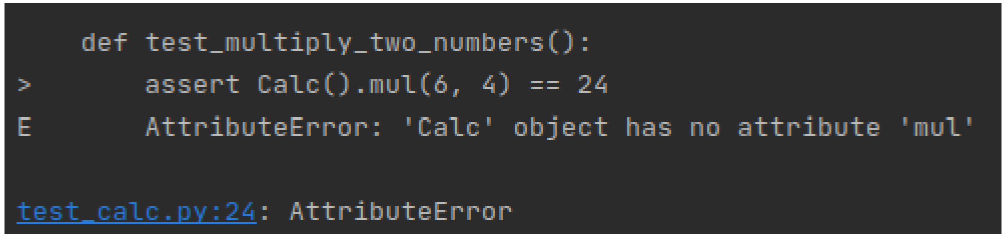
We did all those steps to better understand the approach

Multiplication

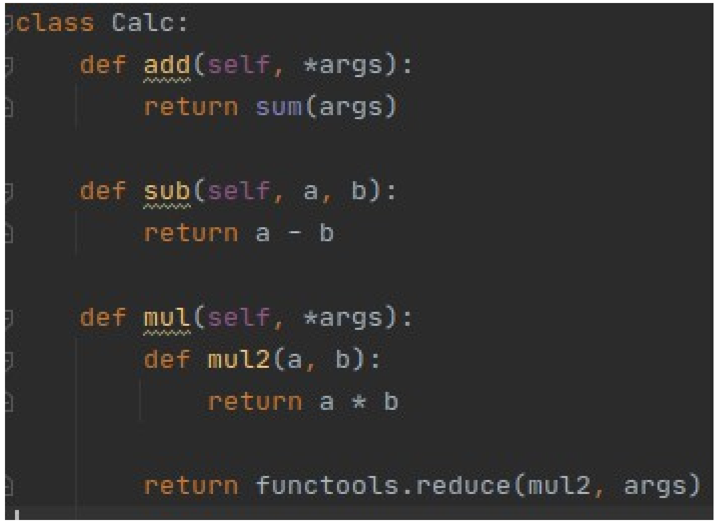
- Similar to addition



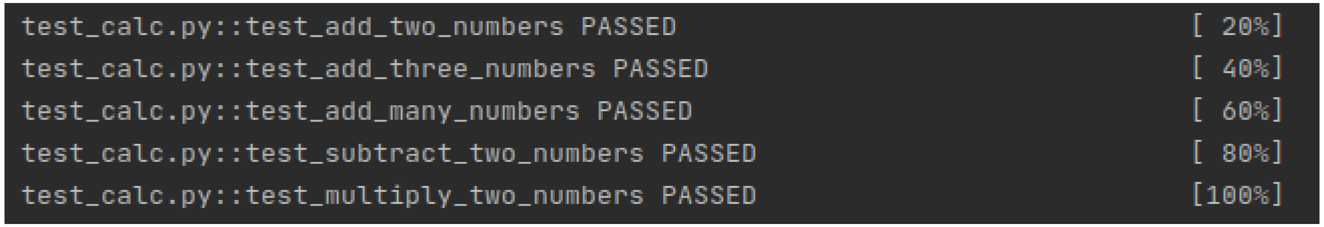
```
def test_multiply_two_numbers():
    assert Calc().mul(6, 4) == 24
```



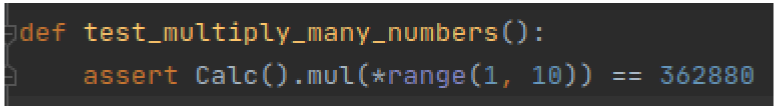
```
    def test_multiply_two_numbers():
>       assert Calc().mul(6, 4) == 24
E       AttributeError: 'Calc' object has no attribute 'mul'

test_calc.py:24: AttributeError
```



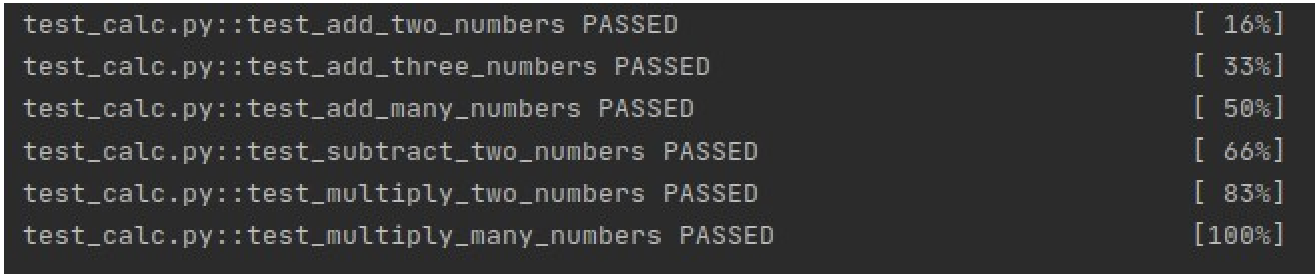
```
class Calc:
    def add(self, *args):
        return sum(args)

    def sub(self, a, b):
        return a - b

    def mul(self, *args):
        def mul2(a, b):
            return a * b

        return functools.reduce(mul2, args)
```



```
test_calc.py::test_add_two_numbers PASSED                                        [ 20%]
test_calc.py::test_add_three_numbers PASSED                                      [ 40%]
test_calc.py::test_add_many_numbers PASSED                                       [ 60%]
test_calc.py::test_subtract_two_numbers PASSED                                   [ 80%]
test_calc.py::test_multiply_two_numbers PASSED                                   [100%]
```



```
def test_multiply_many_numbers():
    assert Calc().mul(*range(1, 10)) == 362880
```



```
test_calc.py::test_add_two_numbers PASSED                                 [ 16%]
test_calc.py::test_add_three_numbers PASSED                               [ 33%]
test_calc.py::test_add_many_numbers PASSED                                [ 50%]
test_calc.py::test_subtract_two_numbers PASSED                            [ 66%]
test_calc.py::test_multiply_two_numbers PASSED                            [ 83%]
test_calc.py::test_multiply_many_numbers PASSED                           [100%]
```

A non-failing new test… should we keep it?

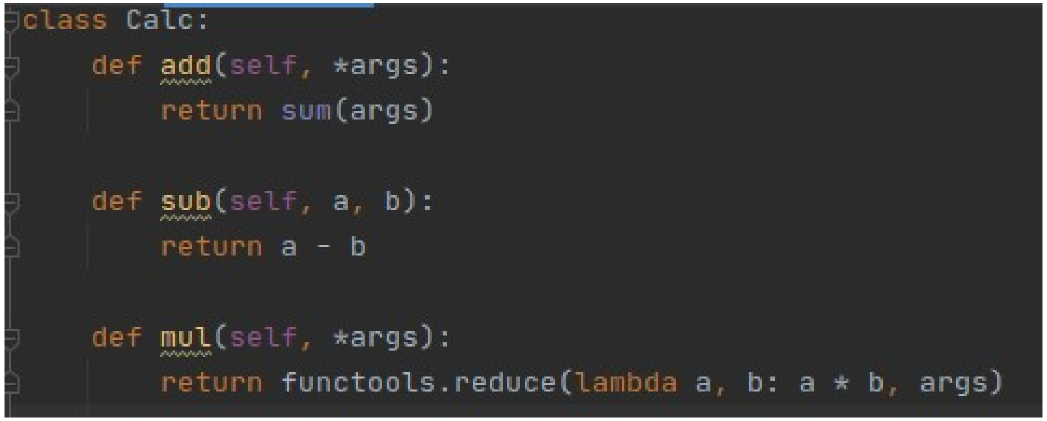
Well, it tests for multiple arguments, so in this case yes, let’s keep it

Usually, new tests must be failing. If not, ask yourself if the new test makes sense

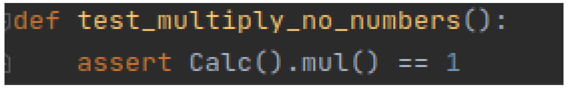
Refactoring

- If all tests pass, we can refactor
- Do not refactor without tests
  - How can you be confidend that you are not breaking something?
  - Better to write tests before refactoring
  - Better with TDD:

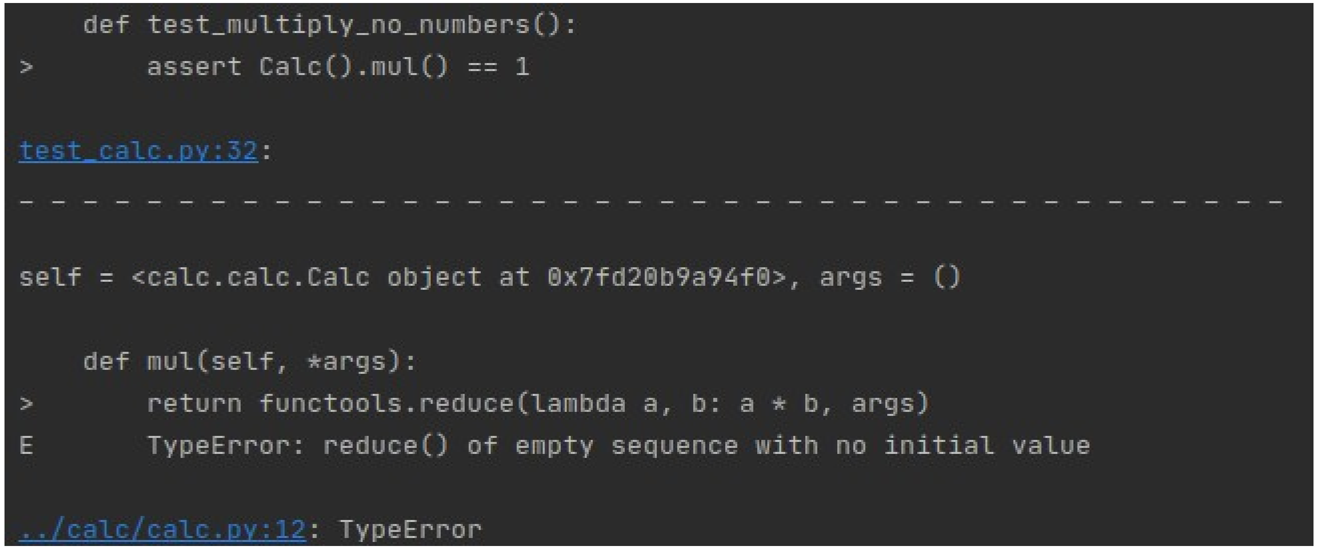
    no tests, no code



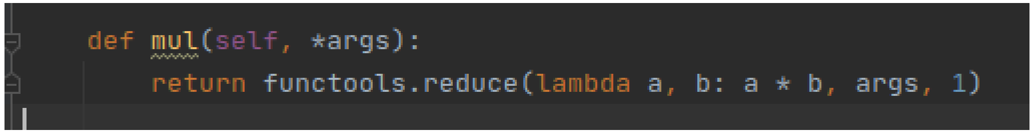
```
class Calc:
    def add(self, *args):
        return sum(args)

    def sub(self, a, b):
        return a - b

    def mul(self, *args):
        return functools.reduce(lambda a, b: a * b, args)
```



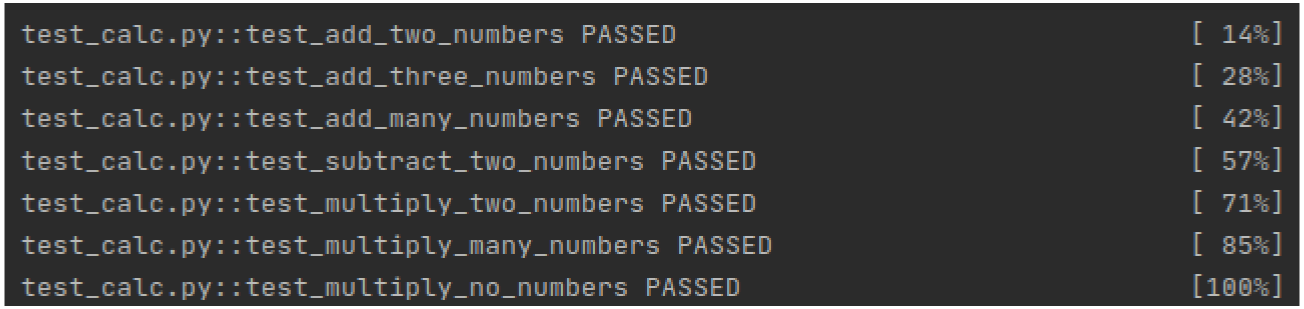
```
def test_multiply_no_numbers():
    assert Calc().mul() == 1
```



```
    def test_multiply_no_numbers():
>       assert Calc().mul() == 1

test_calc.py:32:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

self = <calc.calc.Calc object at 0x7fd20b9a94f0>, args = ()

    def mul(self, *args):
>       return functools.reduce(lambda a, b: a * b, args)
E       TypeError: reduce() of empty sequence with no initial value

../calc/calc.py:12: TypeError
```



```
    def mul(self, *args):
        return functools.reduce(lambda a, b: a * b, args, 1)
```

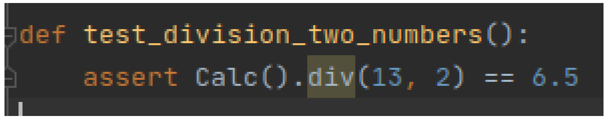


```
test_calc.py::test_add_two_numbers PASSED                          [ 14%]
test_calc.py::test_add_three_numbers PASSED                        [ 28%]
test_calc.py::test_add_many_numbers PASSED                         [ 42%]
test_calc.py::test_subtract_two_numbers PASSED                     [ 57%]
test_calc.py::test_multiply_two_numbers PASSED                     [ 71%]
test_calc.py::test_multiply_many_numbers PASSED                    [ 85%]
test_calc.py::test_multiply_no_numbers PASSED                      [100%]
```

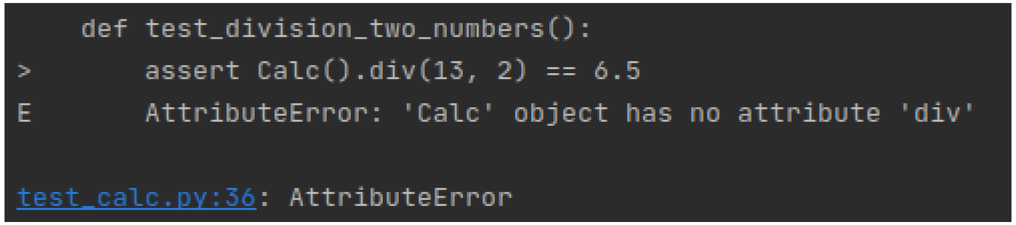
Always tests boundary cases

Division

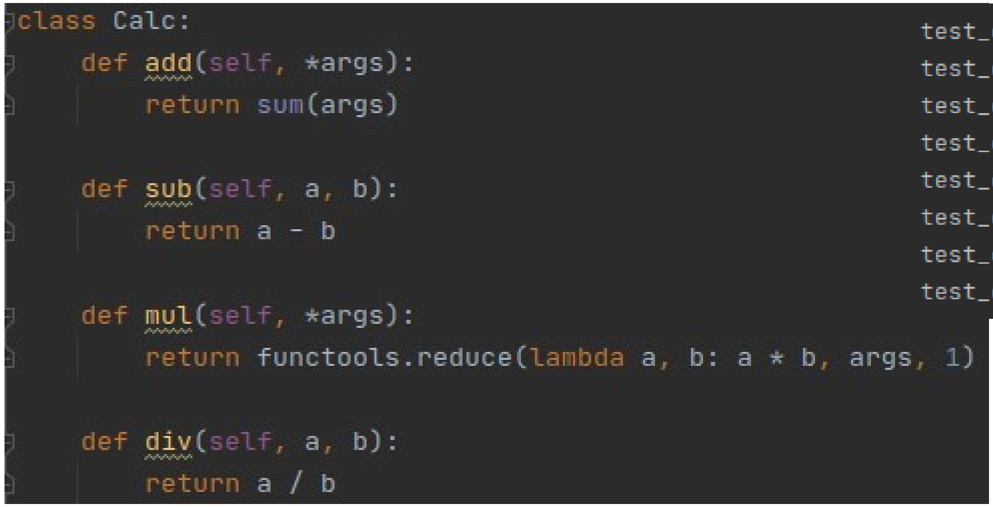
- There must be a division function returing a float



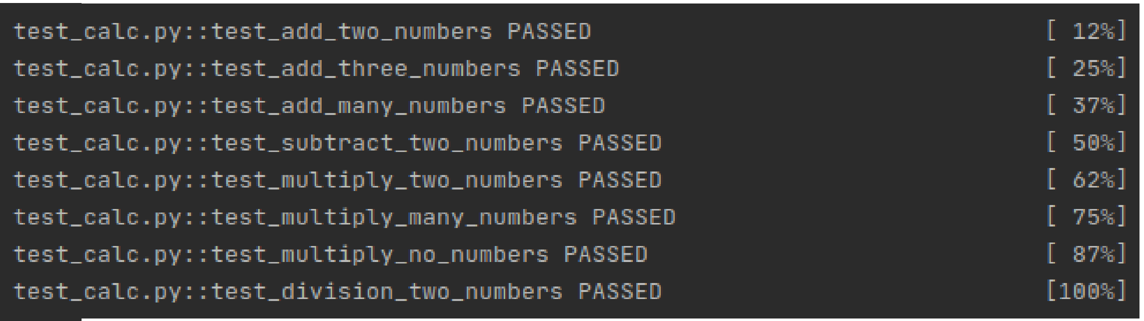
```
def test_division_two_numbers():
    assert Calc().div(13, 2) == 6.5
```



```
    def test_division_two_numbers():
>       assert Calc().div(13, 2) == 6.5
E       AttributeError: 'Calc' object has no attribute 'div'

test_calc.py:36: AttributeError
```



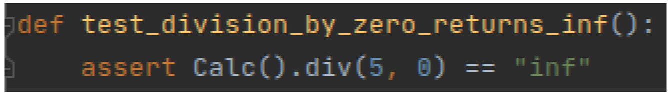
```
class Calc:
    def add(self, *args):
        return sum(args)

    def sub(self, a, b):
        return a - b

    def mul(self, *args):
        return functools.reduce(lambda a, b: a * b, args, 1)

    def div(self, a, b):
        return a / b
```



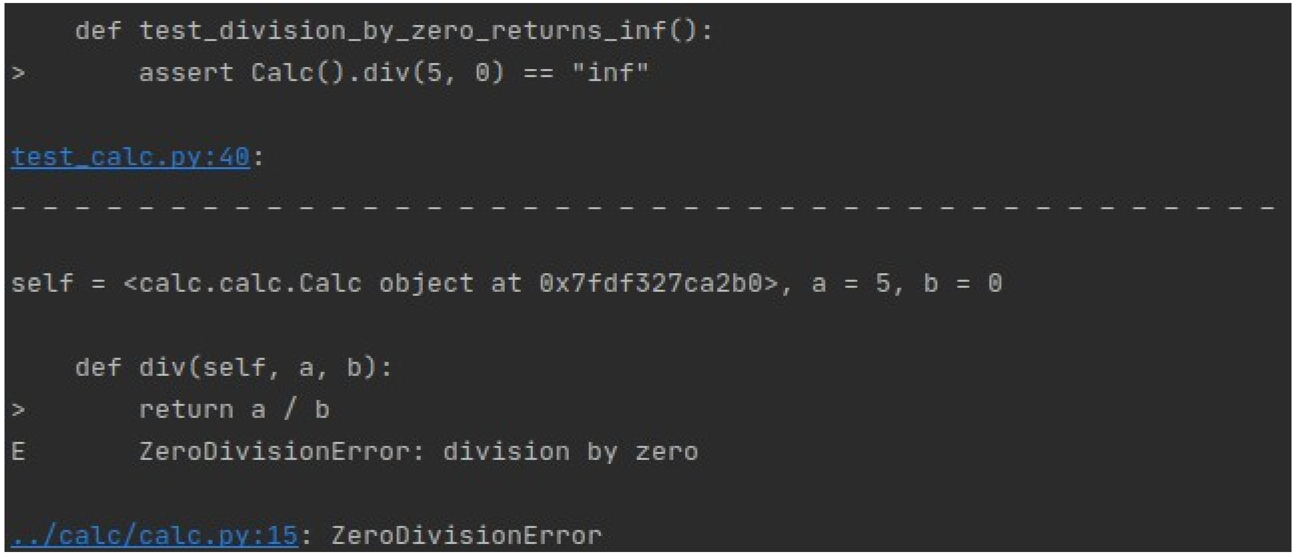
```
test_calc.py::test_add_two_numbers PASSED                  [ 12%]
test_calc.py::test_add_three_numbers PASSED                [ 25%]
test_calc.py::test_add_many_numbers PASSED                 [ 37%]
test_calc.py::test_subtract_two_numbers PASSED             [ 50%]
test_calc.py::test_multiply_two_numbers PASSED             [ 62%]
test_calc.py::test_multiply_many_numbers PASSED            [ 75%]
test_calc.py::test_multiply_no_numbers PASSED              [ 87%]
test_calc.py::test_division_two_numbers PASSED             [100%]
```

Requirement: division by zero shall return the string "inf"

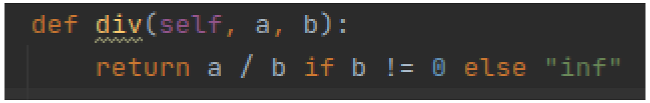
Very strange, but it is just an example



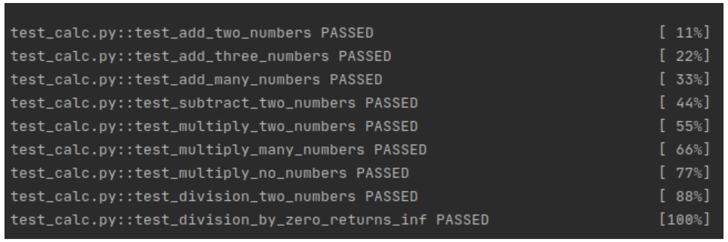
```
def test_division_by_zero_returns_inf():
    assert Calc().div(5, 0) == "inf"
```



```
    def test_division_by_zero_returns_inf():
>       assert Calc().div(5, 0) == "inf"

test_calc.py:40:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

self = <calc.calc.Calc object at 0x7fdf327ca2b0>, a = 5, b = 0

    def div(self, a, b):
>       return a / b
E       ZeroDivisionError: division by zero

../calc/calc.py:15: ZeroDivisionError
```



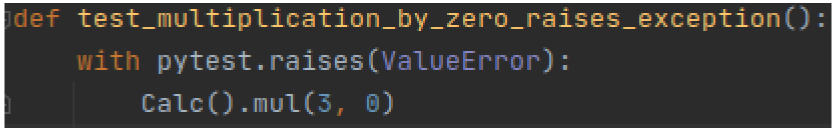
```
def div(self, a, b):
    return a / b if b != 0 else "inf"
```



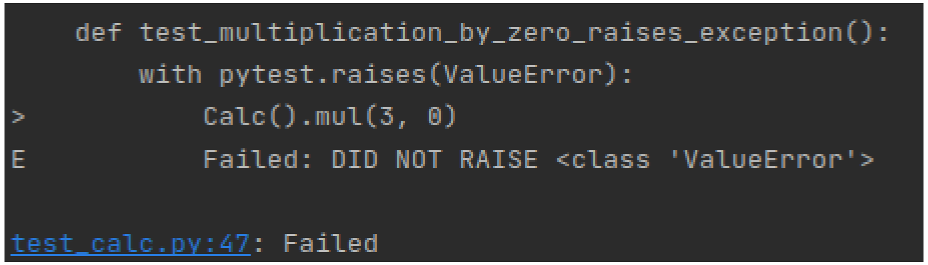
```
test_calc.py::test_add_two_numbers PASSED                              [ 11%]
test_calc.py::test_add_three_numbers PASSED                            [ 22%]
test_calc.py::test_add_many_numbers PASSED                             [ 33%]
test_calc.py::test_subtract_two_numbers PASSED                         [ 44%]
test_calc.py::test_multiply_two_numbers PASSED                         [ 55%]
test_calc.py::test_multiply_many_numbers PASSED                        [ 66%]
test_calc.py::test_multiply_no_numbers PASSED                          [ 77%]
test_calc.py::test_division_two_numbers PASSED                         [ 88%]
test_calc.py::test_division_by_zero_returns_inf PASSED                 [100%]
```

Testing exceptions

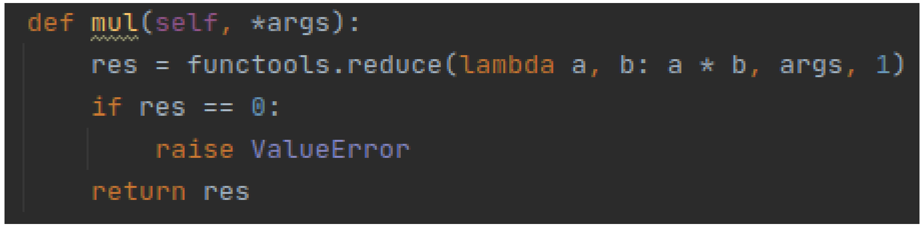
- Requirement: multiplication by zero must raise a ValueError exception
- We can use pytest.raises to assert exception raising



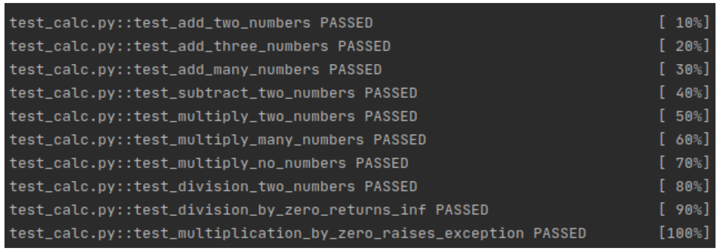
```python
def test_multiplication_by_zero_raises_exception():
    with pytest.raises(ValueError):
        Calc().mul(3, 0)
```



```
    def test_multiplication_by_zero_raises_exception():
        with pytest.raises(ValueError):
>           Calc().mul(3, 0)
E           Failed: DID NOT RAISE <class 'ValueError'>

test_calc.py:47: Failed
```



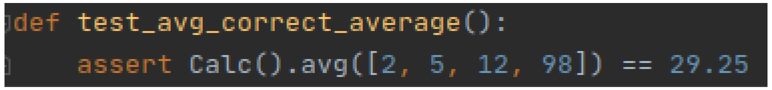
```python
def mul(self, *args):
    res = functools.reduce(lambda a, b: a * b, args, 1)
    if res == 0:
        raise ValueError
    return res
```



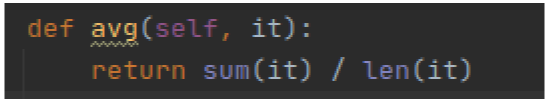
```
test_calc.py::test_add_two_numbers PASSED                                    [ 10%]
test_calc.py::test_add_three_numbers PASSED                                  [ 20%]
test_calc.py::test_add_many_numbers PASSED                                   [ 30%]
test_calc.py::test_subtract_two_numbers PASSED                               [ 40%]
test_calc.py::test_multiply_two_numbers PASSED                               [ 50%]
test_calc.py::test_multiply_many_numbers PASSED                              [ 60%]
test_calc.py::test_multiply_no_numbers PASSED                                [ 70%]
test_calc.py::test_division_two_numbers PASSED                               [ 80%]
test_calc.py::test_division_by_zero_returns_inf PASSED                       [ 90%]
test_calc.py::test_multiplication_by_zero_raises_exception PASSED            [100%]
```

A more complex set of requirements

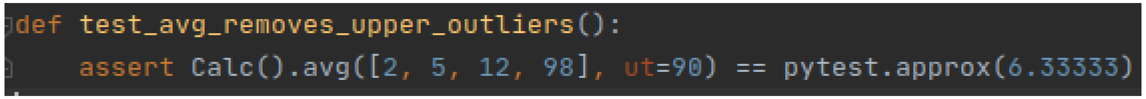
- A function to compute the average of an iterable
- This function shall accept two optional upper and lower thresholds to remove outliers
- Let’s break to simple tests
  - The function accepts an iterable and computes the average, eg. avg([2, 5, 12, 98]) == 29.25
  - The function accepts an optional upper threshold, eg. avg([2, 5, 12, 98], ut=90) == avg([2, 5, 12])
  - The function accepts an optional lower threshold, eg. avg([2, 5, 12, 98], lt=10) == avg([12, 98])
  - The upper threshold stays in, eg. avg([2, 5, 12, 98], ut=98) == avg([2, 5, 12, 98])
  - The lower threshold stays in, eg. avg([2, 5, 12, 98], lt=5) == avg([5, 12, 98])
  - The function works with an empty list, eg. avg([]) == None
  - The function works if the list is empty after outlier removal, eg. avg([12, 98], lt=15, ut=90) == None
  - The function outlier removal works if the list is empty, eg. avg([], lt=15, ut=90) == None



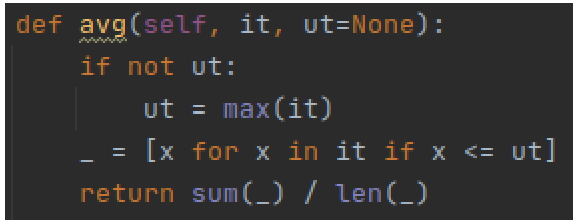
```python
def test_avg_correct_average():
    assert Calc().avg([2, 5, 12, 98]) == 29.25
```



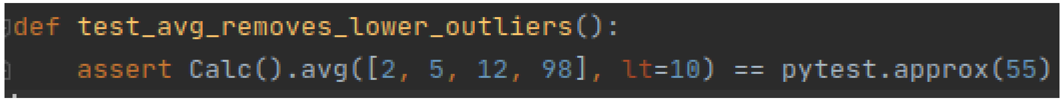
```python
def avg(self, it):
    return sum(it) / len(it)
```



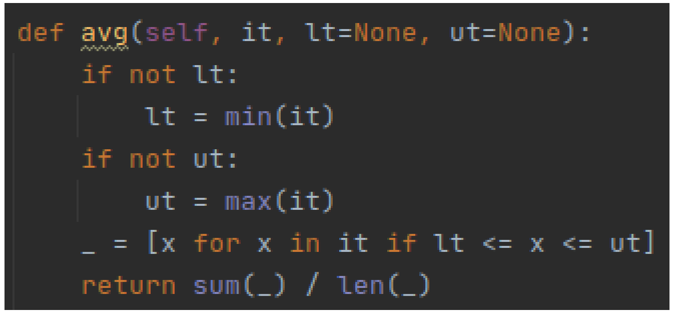
```python
def test_avg_removes_upper_outliers():
    assert Calc().avg([2, 5, 12, 98], ut=90) == pytest.approx(6.33333)
```



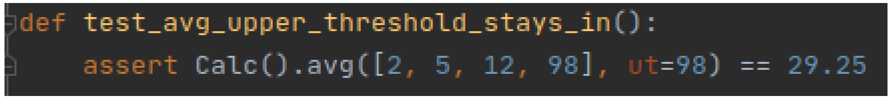
```python
def avg(self, it, ut=None):
    if not ut:
        ut = max(it)
    _ = [x for x in it if x <= ut]
    return sum(_) / len(_)
```



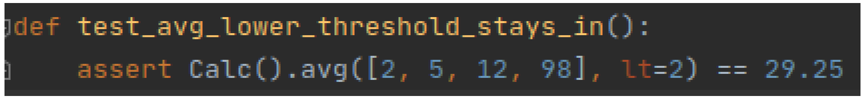
```python
def test_avg_removes_lower_outliers():
    assert Calc().avg([2, 5, 12, 98], lt=10) == pytest.approx(55)
```



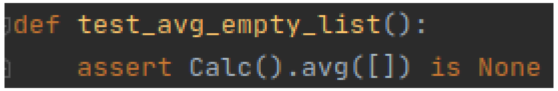
```python
def avg(self, it, lt=None, ut=None):
    if not lt:
        lt = min(it)
    if not ut:
        ut = max(it)
    _ = [x for x in it if lt <= x <= ut]
    return sum(_) / len(_)
```



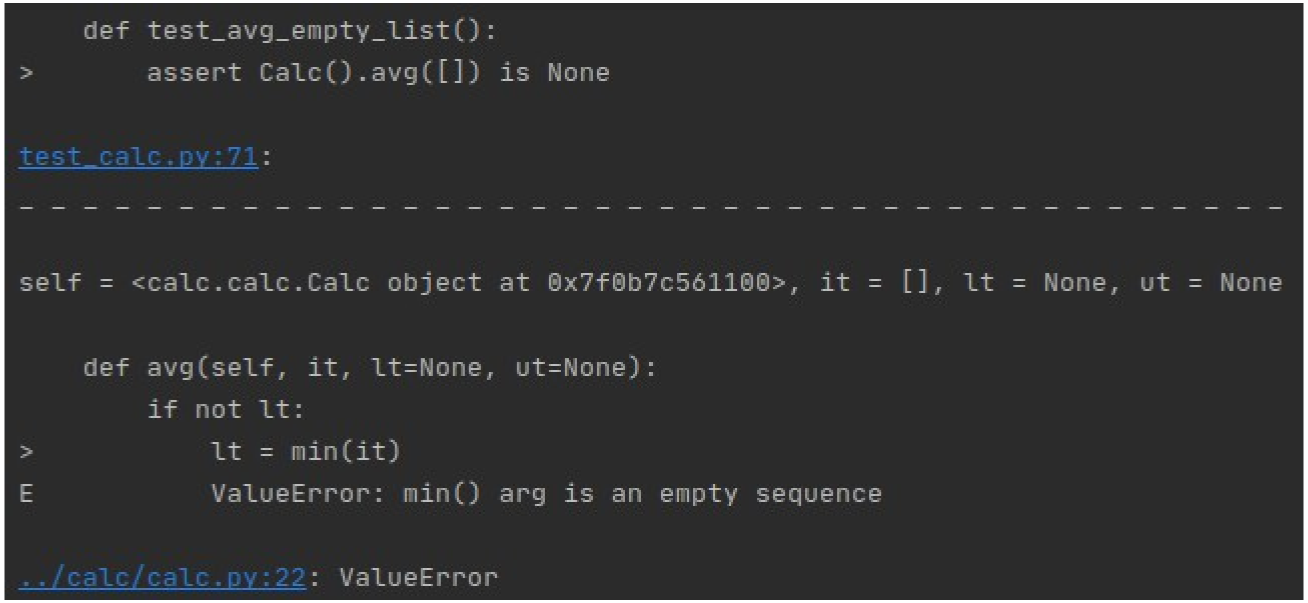
```python
def test_avg_upper_threshold_stays_in():
    assert Calc().avg([2, 5, 12, 98], ut=98) == 29.25
```



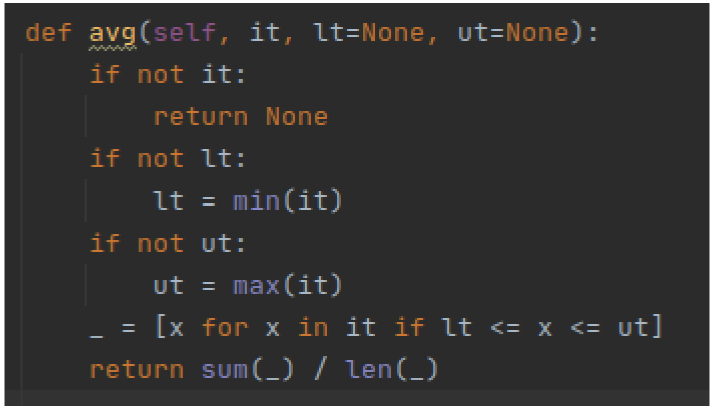
```python
def test_avg_lower_threshold_stays_in():
    assert Calc().avg([2, 5, 12, 98], lt=2) == 29.25
```



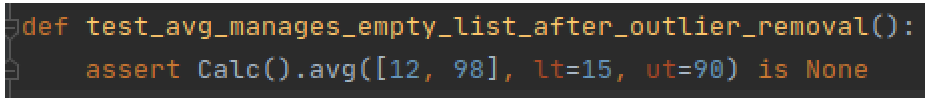
```
def test_avg_empty_list():
    assert Calc().avg([]) is None
```



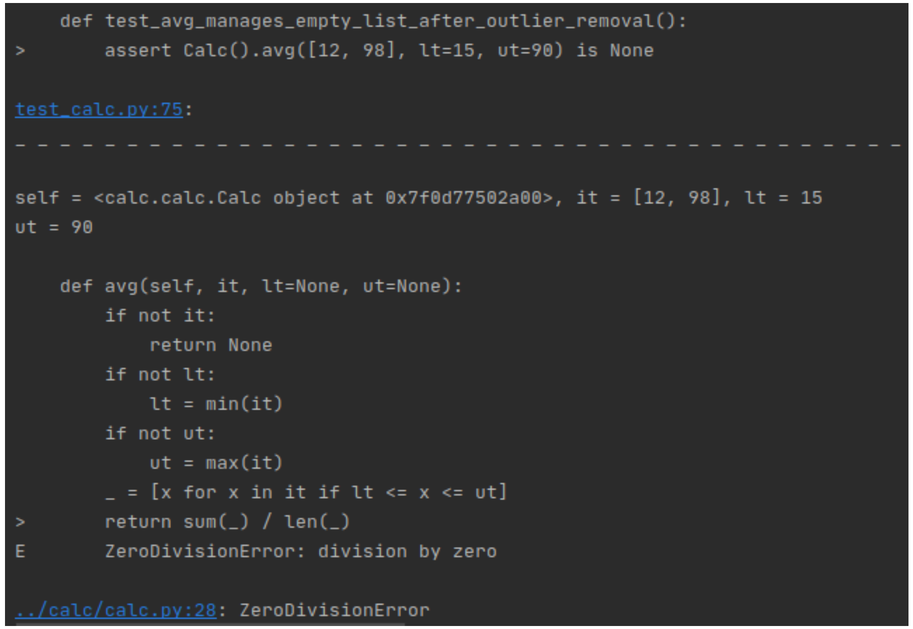
```
    def test_avg_empty_list():
>       assert Calc().avg([]) is None

test_calc.py:71:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

self = <calc.calc.Calc object at 0x7f0b7c561100>, it = [], lt = None, ut = None

    def avg(self, it, lt=None, ut=None):
        if not lt:
>           lt = min(it)
E           ValueError: min() arg is an empty sequence

../calc/calc.py:22: ValueError
```



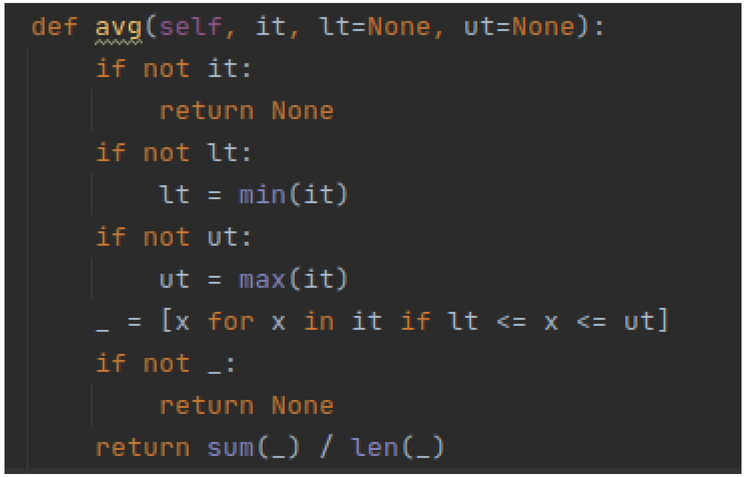
```
def avg(self, it, lt=None, ut=None):
    if not it:
        return None
    if not lt:
        lt = min(it)
    if not ut:
        ut = max(it)
    _ = [x for x in it if lt <= x <= ut]
    return sum(_) / len(_)
```



```python
def test_avg_manages_empty_list_after_outlier_removal():
    assert Calc().avg([12, 98], lt=15, ut=90) is None
```

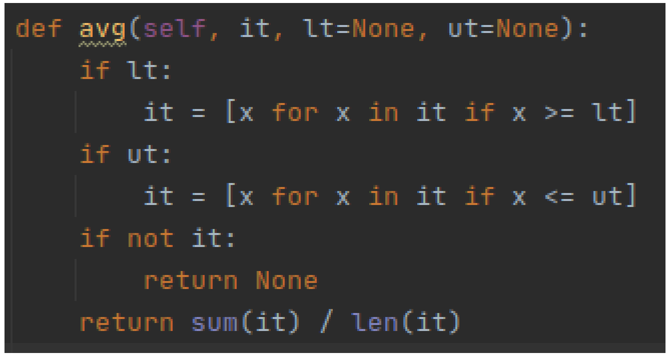


```
    def test_avg_manages_empty_list_after_outlier_removal():
>       assert Calc().avg([12, 98], lt=15, ut=90) is None

test_calc.py:75:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

self = <calc.calc.Calc object at 0x7f0d77502a00>, it = [12, 98], lt = 15
ut = 90

    def avg(self, it, lt=None, ut=None):
        if not it:
            return None
        if not lt:
            lt = min(it)
        if not ut:
            ut = max(it)
        _ = [x for x in it if lt <= x <= ut]
>       return sum(_) / len(_)
E       ZeroDivisionError: division by zero

../calc/calc.py:28: ZeroDivisionError
```

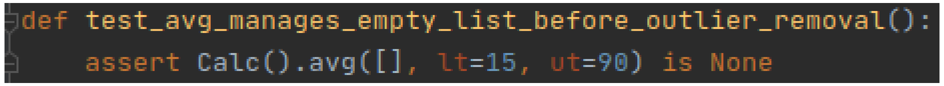


```python
def avg(self, it, lt=None, ut=None):
    if not it:
        return None
    if not lt:
        lt = min(it)
    if not ut:
        ut = max(it)
    _ = [x for x in it if lt <= x <= ut]
    if not _:
        return None
    return sum(_) / len(_)
```

All test pass. Refactoring time!



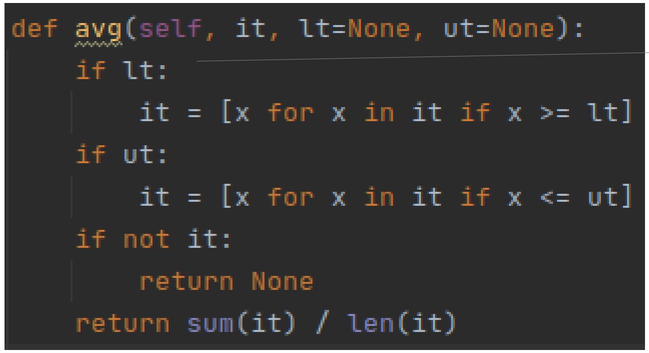
```python
def avg(self, it, lt=None, ut=None):
    if lt:
        it = [x for x in it if x >= lt]
    if ut:
        it = [x for x in it if x <= ut]
    if not it:
        return None
    return sum(it) / len(it)
```



```
def test_avg_manages_empty_list_before_outlier_removal():
    assert Calc().avg([], lt=15, ut=90) is None
```

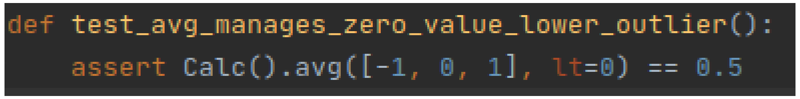
Tests from bug reports

- A bug is an example of a missing test in your suite

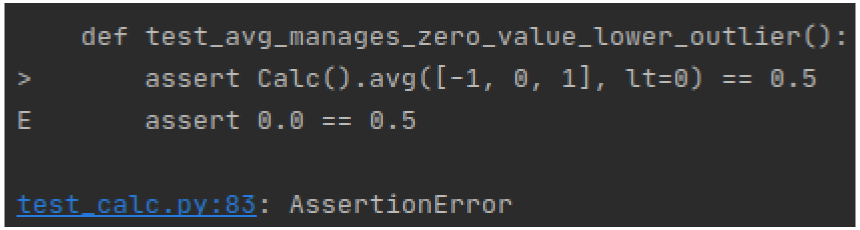


```python
def avg(self, it, lt=None, ut=None):
    if lt:
        it = [x for x in it if x >= lt]
    if ut:
        it = [x for x in it if x <= ut]
    if not it:
        return None
    return sum(it) / len(it)
```

What if lt is 0?

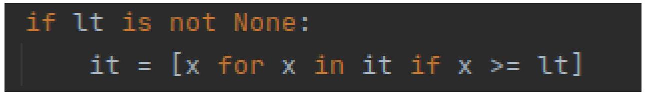


```python
def test_avg_manages_zero_value_lower_outlier():
    assert Calc().avg([-1, 0, 1], lt=0) == 0.5
```



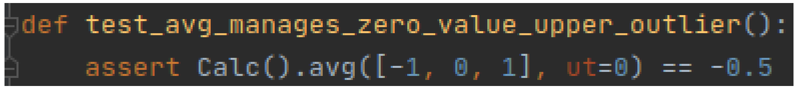
```
    def test_avg_manages_zero_value_lower_outlier():
>       assert Calc().avg([-1, 0, 1], lt=0) == 0.5
E       assert 0.0 == 0.5

test_calc.py:83: AssertionError
```

Fix

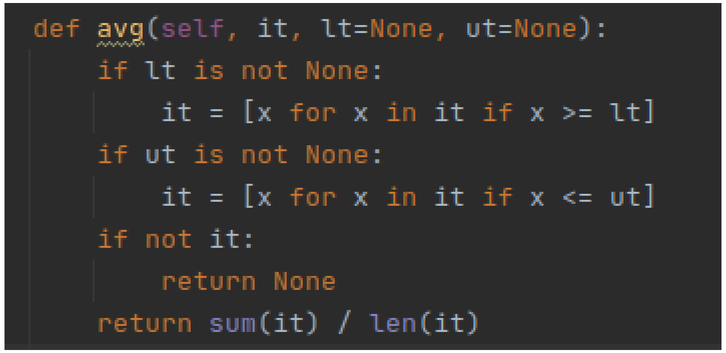


```python
if lt is not None:
    it = [x for x in it if x >= lt]
```

Same problem for ut



```
def test_avg_manages_zero_value_upper_outlier():
    assert Calc().avg([-1, 0, 1], ut=0) == -0.5
```



```
def avg(self, it, lt=None, ut=None):
    if lt is not None:
        it = [x for x in it if x >= lt]
    if ut is not None:
        it = [x for x in it if x <= ut]
    if not it:
        return None
    return sum(it) / len(it)
```

Note that we are refactoring… luckily we have regression tests!

Last thing, but not least, let’s try to run tests with coverage analysis

Lines of code that are not covered by tests are either unreachable or witness missing tests

Summing up

1) Test first, code later
2) Add the bare minimum amount of code you need to pass the tests
3) You shouldn’t have more than one failing test at a time
4) Write code that passes the tests. Then refactor it.
5) A test should fail the first time you run it. If it doesn’t, ask yourself why you are adding it.
6) Never refactor without tests

Questions