Unit Testing

Giuseppe Scanniello

Simone Romano

Michelangelo Esposito

Terminology

Test: The act of exercising a software system with a test case
- Its goal is to break systems or demonstrate their correct execution

Test case: a recognized work product
- It is associated to a system behavior
- It has a set of inputs and expected outputs (i.e., **oracle**)

Test suite: Set of test cases

Unit testing

A way of answering "Do the parts of my system perform correctly alone?"

Level of testing where units (or components) are individually tested

Unit: minimal part of a system that can be tested in isolation

In **procedural** software systems a **unit** is:

- Procedure, function, body of code that implements a single feature, …

In **OO** software systems a **unit** is:

- Class, method, …

How to design (unit) tests?

Happy path testing:

Tests for the correct usage of the software system

Unhappy path testing:

Tests for an incorrect usage of the software system

How to execute them?

- Executing tests should be automatic
  - Design once, execute many times
  - Need of **test automation**

Test automation

The use of software to perform or support test activities like test execution and results checking

Benefits:

- **Cost reduction**---the tester should just press a button to start executing tests
- **Human error reduction**---providing input values and checking outcomes manually is an error-prone activity
- **Foster regression testing**---repeatable tests allow easily testing a system after modifications to it

unittest library

A test automation module (in particular, a unit testing module)

Included in the Python standard library

Similar (xUnit) frameworks are available for other languages

xUnit frameworks

CUnit---C

CppUnit---C++

COBOLUnit---Cobol

DUnit---Delphy

FUnit---Fortran

JUnit --- Java

JSUnit---JavaScript

mlUnit---Matlab

NUnit---.Net

PHPUnit---PHP

SUnit---Smalltalk

SimplyVBUnit---Visual Basic

XCTest---Xcode

…

Basic concepts

Assertion, it verifies a single expected result

Test method, it verifies a single system behavior---it corresponds to a test case

Test class, it embodies all test methods for a given class---it can be seen as a test suite

Assertion

A method called by a test method to verify a single expected result

`assertTrue`	`assertFalse`
`assertIsNone`	`assertNotNone`
`assertEqual`	`assertIsInstance`
`assertRegexpMatches`	

Assertion

Examples:

```
assertTrue(a)
assertEqual(0,a)
assertIsInstance(cls,obj)
```

Cause a test method to fail if `a!= true`

Cause a test method to fail if `a!=0`

Cause a test method to fail if `obj` is not of type `cls`

Expected output

Actual output

Assertion

All assertions accept a `string` argument (in the last position) to describe the reason of a failure

Examples:

```
assertTrue(a, “value is not True”);
assertEquals(0, a, “value not 0”);
```

Test class

A class that embodies a set of test methods and related code

Usually, there is a test class for each system class

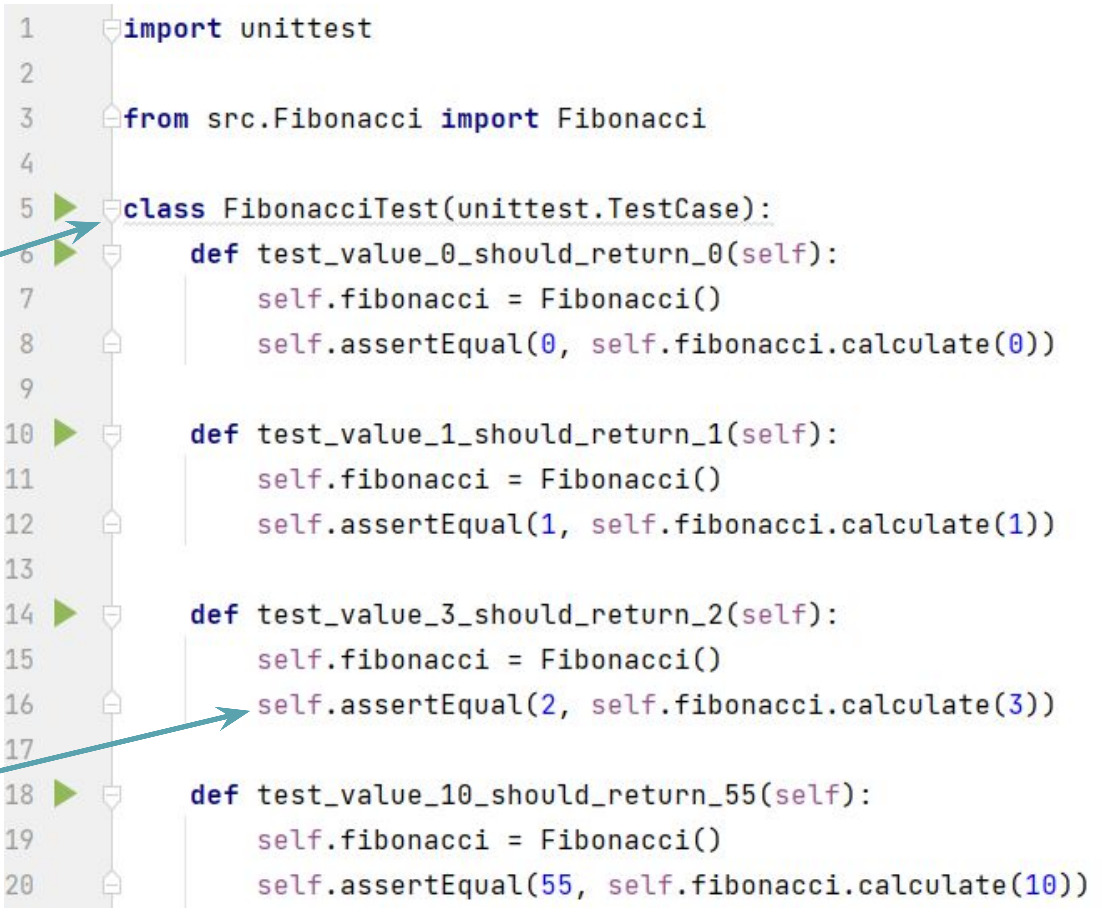
A test class must inherit from unittest.TestCase

Typical test class structure

Typically, the test class name for a class "A" is "ATest" (but not always)

Test class must inherit from unittest.TestCase

Assertion (one of several kinds)



```python
import unittest

from src.Fibonacci import Fibonacci

class FibonacciTest(unittest.TestCase):
    def test_value_0_should_return_0(self):
        self.fibonacci = Fibonacci()
        self.assertEqual(0, self.fibonacci.calculate(0))

    def test_value_1_should_return_1(self):
        self.fibonacci = Fibonacci()
        self.assertEqual(1, self.fibonacci.calculate(1))

    def test_value_3_should_return_2(self):
        self.fibonacci = Fibonacci()
        self.assertEqual(2, self.fibonacci.calculate(3))

    def test_value_10_should_return_55(self):
        self.fibonacci = Fibonacci()
        self.assertEqual(55, self.fibonacci.calculate(10))
```

Test method

Test method names MUST start with “test_”:

```
def test_<something>: ...
```

Test method names should be meaningful; they should provide a clear idea what the test is for

Test method

It can contain any code:

- Local variables, calculations
- Call to helper methods
- …

In particular, it should always contain at least one **assertion**

Test method

Example:

```
def test_list_size(self):
    list = []
    list.append(1)
    list.append(2)
    list.append(3)
    self.assertEqual(3, len(list))
```

Fibonacci example

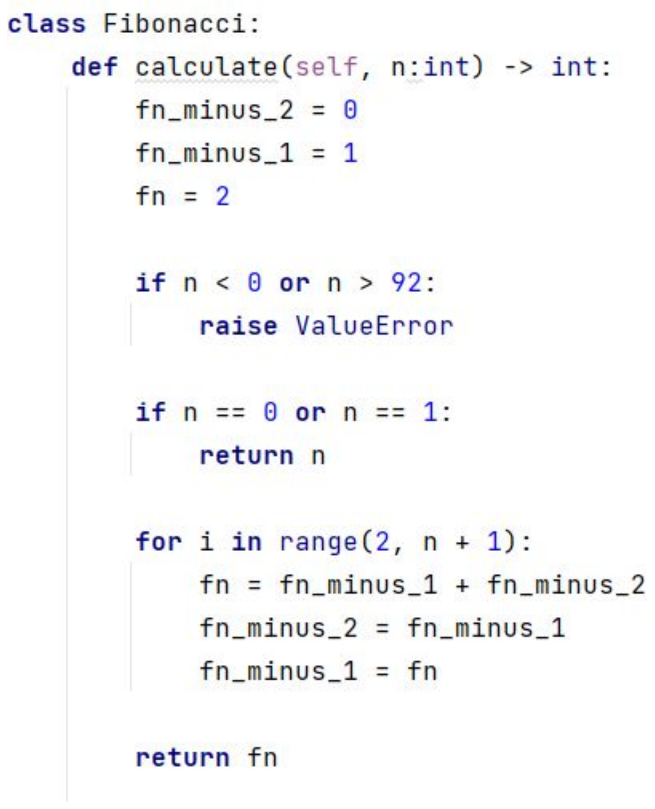
Let us apply these basic concepts to the Fibonacci example

The Fibonacci number (f_n) is defined as follows:

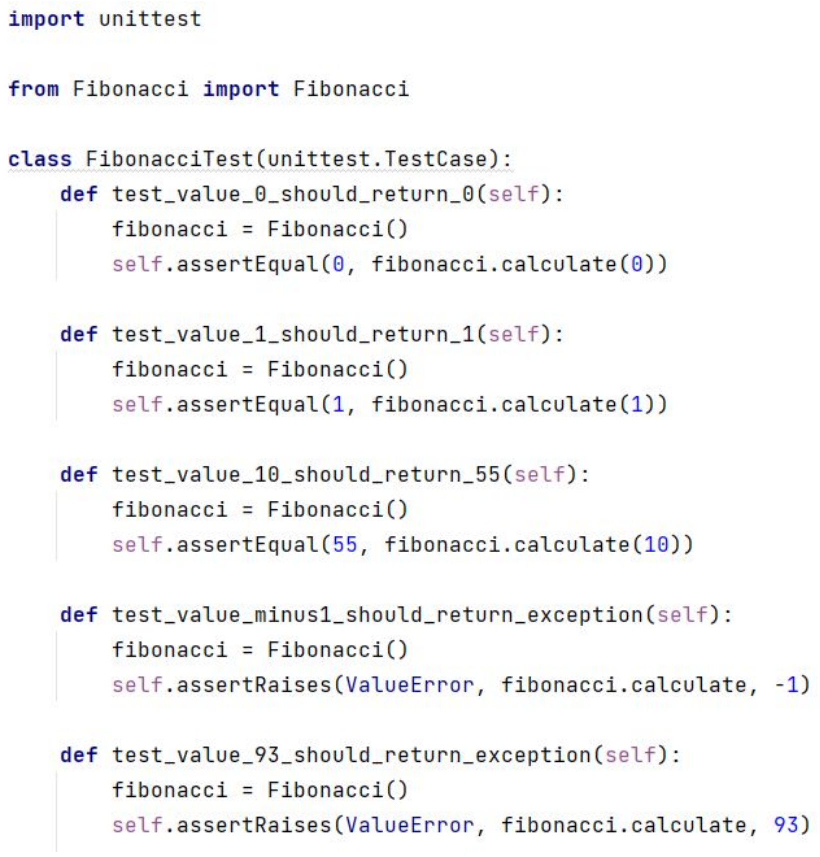
$$\boxed{\begin{gathered} f_0 = 0 \; if \; n = 0 \\ f_1 = 1 \; if \; n = 1 \\ f_n = f_{n-1} + f_{n-2} \; if \; n > 1 \end{gathered}}$$

We use **PyCharm IDE** in conjunction to the **unittest** module

Production and test code for the Fibonacci example



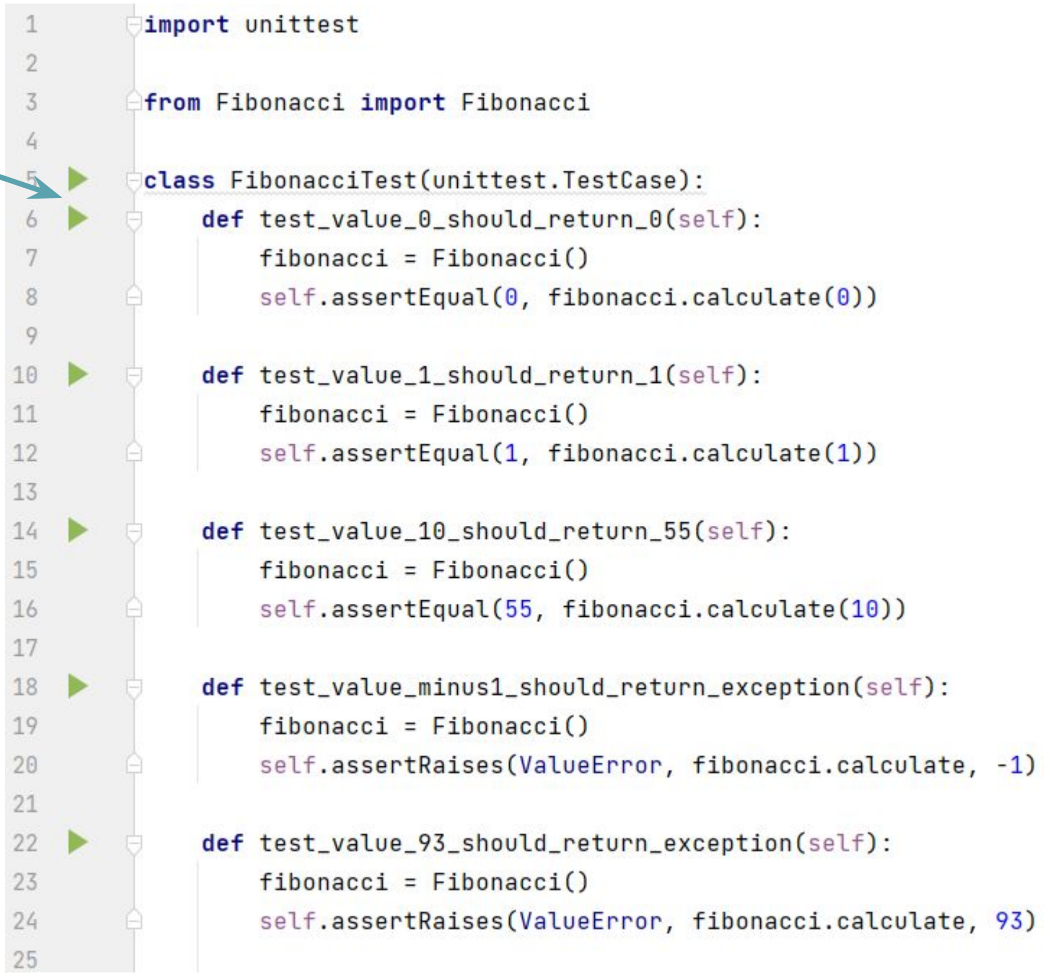
```python
class Fibonacci:
    def calculate(self, n:int) -> int:
        fn_minus_2 = 0
        fn_minus_1 = 1
        fn = 2

        if n < 0 or n > 92:
            raise ValueError

        if n == 0 or n == 1:
            return n

        for i in range(2, n + 1):
            fn = fn_minus_1 + fn_minus_2
            fn_minus_2 = fn_minus_1
            fn_minus_1 = fn

        return fn
```



```python
import unittest

from Fibonacci import Fibonacci

class FibonacciTest(unittest.TestCase):
    def test_value_0_should_return_0(self):
        fibonacci = Fibonacci()
        self.assertEqual(0, fibonacci.calculate(0))

    def test_value_1_should_return_1(self):
        fibonacci = Fibonacci()
        self.assertEqual(1, fibonacci.calculate(1))

    def test_value_10_should_return_55(self):
        fibonacci = Fibonacci()
        self.assertEqual(55, fibonacci.calculate(10))

    def test_value_minus1_should_return_exception(self):
        fibonacci = Fibonacci()
        self.assertRaises(ValueError, fibonacci.calculate, -1)

    def test_value_93_should_return_exception(self):
        fibonacci = Fibonacci()
        self.assertRaises(ValueError, fibonacci.calculate, 93)
```

Test execution in PyCharm

Run the whole suite or a single test case with the PyCharm built-in runner

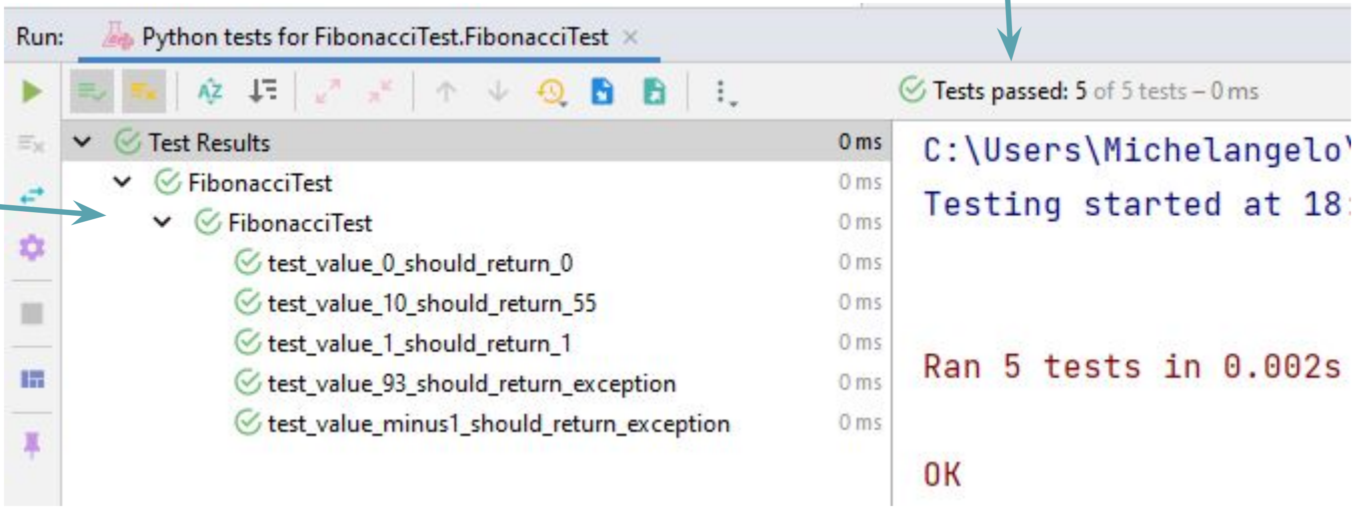


```python
import unittest

from Fibonacci import Fibonacci

class FibonacciTest(unittest.TestCase):
    def test_value_0_should_return_0(self):
        fibonacci = Fibonacci()
        self.assertEqual(0, fibonacci.calculate(0))

    def test_value_1_should_return_1(self):
        fibonacci = Fibonacci()
        self.assertEqual(1, fibonacci.calculate(1))

    def test_value_10_should_return_55(self):
        fibonacci = Fibonacci()
        self.assertEqual(55, fibonacci.calculate(10))

    def test_value_minus1_should_return_exception(self):
        fibonacci = Fibonacci()
        self.assertRaises(ValueError, fibonacci.calculate, -1)

    def test_value_93_should_return_exception(self):
        fibonacci = Fibonacci()
        self.assertRaises(ValueError, fibonacci.calculate, 93)
```

Watch test outcomes

Status

Run tests, results (success, failure, ignore, error) and execution time



Tests passed: 5 of 5 tests – 0 ms

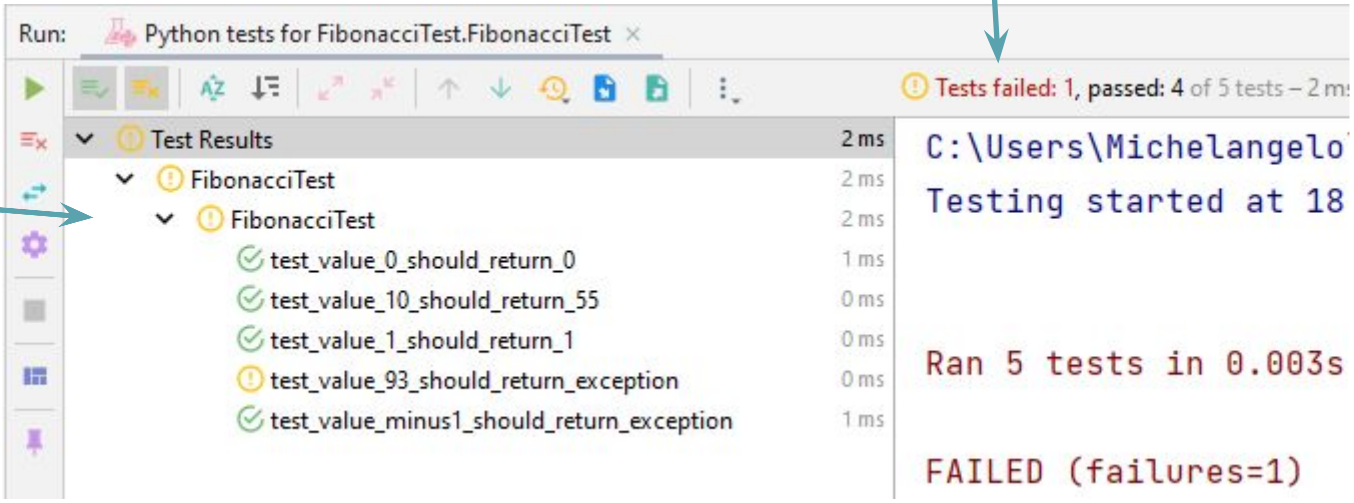
Test Results	0 ms
FibonacciTest	0 ms
FibonacciTest	0 ms
test_value_0_should_return_0	0 ms
test_value_10_should_return_55	0 ms
test_value_1_should_return_1	0 ms
test_value_93_should_return_exception	0 ms
test_value_minus1_should_return_exception	0 ms

```
C:\Users\Michelangelo\
Testing started at 18:


Ran 5 tests in 0.002s

OK
```

Watch test outcomes (if there was a failure)

Status

Run tests, results (success, failure, ignore, error) and execution time



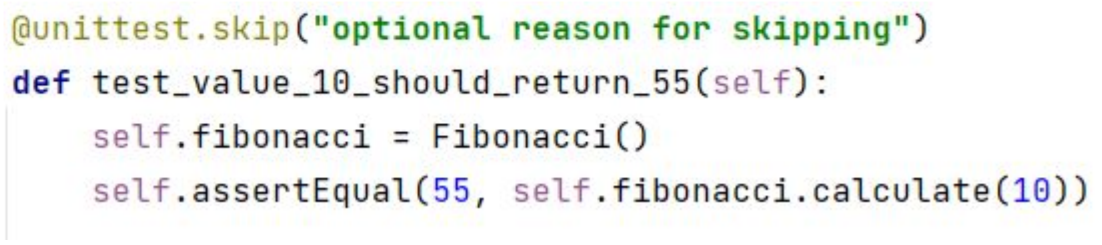
Tests failed: 1, passed: 4 of 5 tests – 2 m

```
Test Results                                      2 ms
  FibonacciTest                                   2 ms
    FibonacciTest                                 2 ms
      test_value_0_should_return_0                1 ms
      test_value_10_should_return_55              0 ms
      test_value_1_should_return_1                0 ms
      test_value_93_should_return_exception       0 ms
      test_value_minus1_should_return_exception   1 ms
```

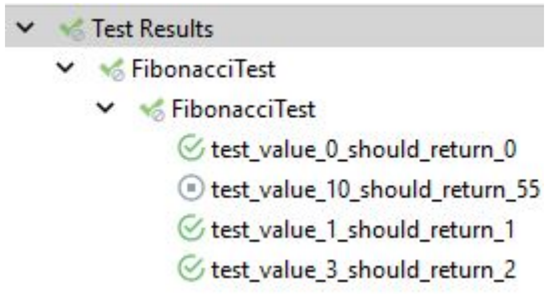
```
C:\Users\Michelangelo
Testing started at 18


Ran 5 tests in 0.003s


FAILED (failures=1)
```

Skipping test methods

The @unittest.skip decorator before a test method allows to skip it during the test execution

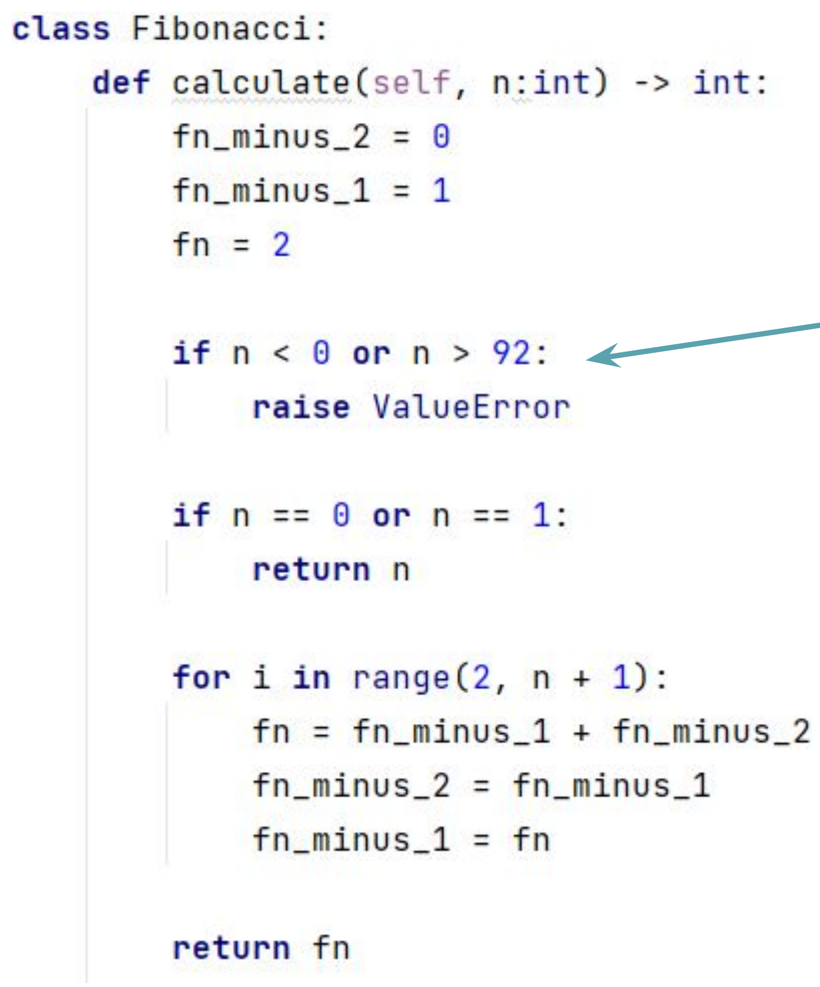


```python
@unittest.skip("optional reason for skipping")
def test_value_10_should_return_55(self):
    self.fibonacci = Fibonacci()
    self.assertEqual(55, self.fibonacci.calculate(10))
```



```
Test Results
  FibonacciTest
    FibonacciTest
      test_value_0_should_return_0
      test_value_10_should_return_55
      test_value_1_should_return_1
      test_value_3_should_return_2
```

Testing and exception



```python
class Fibonacci:
    def calculate(self, n:int) -> int:
        fn_minus_2 = 0
        fn_minus_1 = 1
        fn = 2

        if n < 0 or n > 92:
            raise ValueError

        if n == 0 or n == 1:
            return n

        for i in range(2, n + 1):
            fn = fn_minus_1 + fn_minus_2
            fn_minus_2 = fn_minus_1
            fn_minus_1 = fn

        return fn
```

Invalid values (the upper bound is arbitrary)

Testing and exception

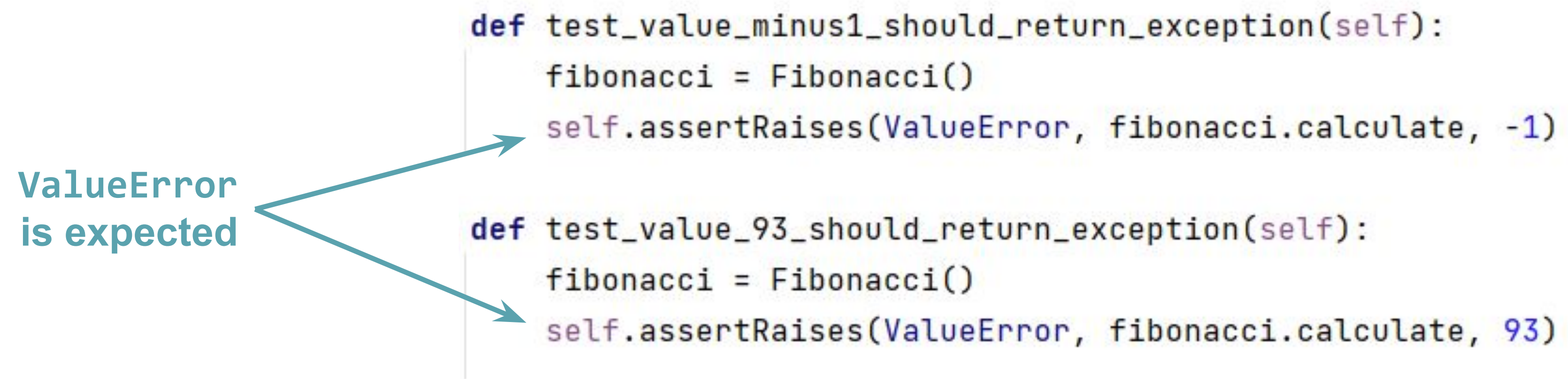
To ensure that a given exception has been raised, the `assertRaises` assertion is used

```
assertRaises(exception_type, method, arguments)
```

In our Fibonacci example:

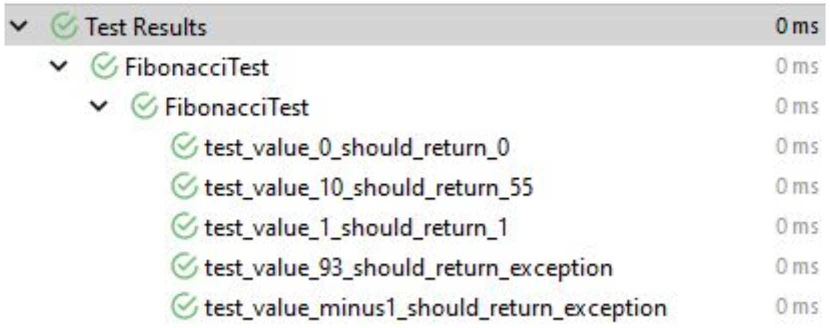
```
assertRaises(ValueError, Fibonacci.calculate, -1)
```

Testing and exception

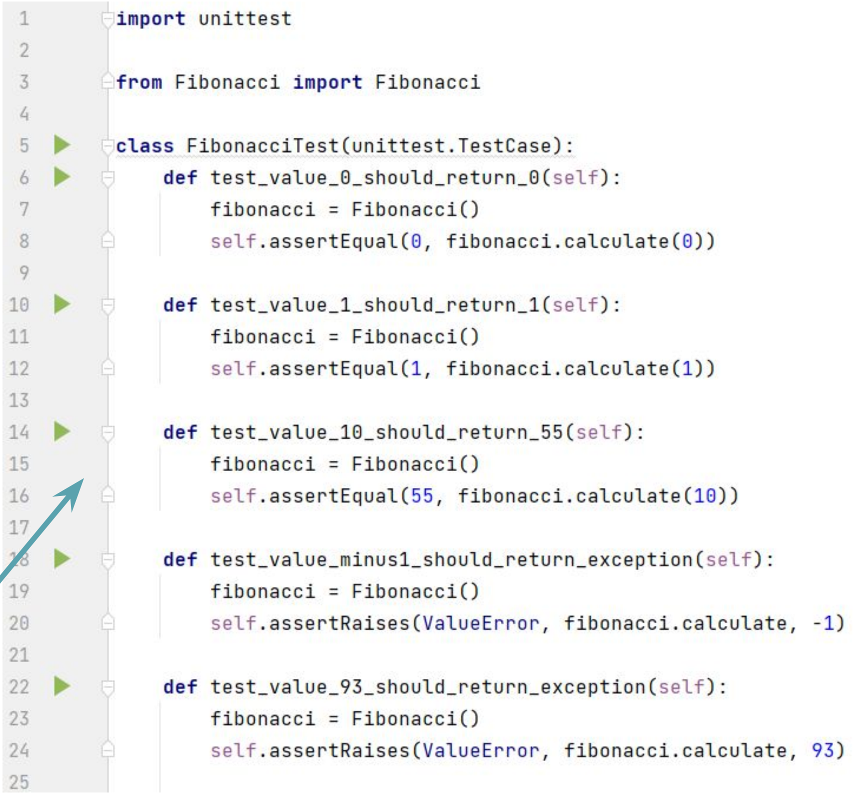


```
def test_value_minus1_should_return_exception(self):
    fibonacci = Fibonacci()
    self.assertRaises(ValueError, fibonacci.calculate, -1)

def test_value_93_should_return_exception(self):
    fibonacci = Fibonacci()
    self.assertRaises(ValueError, fibonacci.calculate, 93)
```

Test execution order



```
Test Results                                   0 ms
  FibonacciTest                                0 ms
    FibonacciTest                              0 ms
      test_value_0_should_return_0             0 ms
      test_value_10_should_return_55           0 ms
      test_value_1_should_return_1             0 ms
      test_value_93_should_return_exception    0 ms
      test_value_minus1_should_return_exception 0 ms
```



```python
import unittest

from Fibonacci import Fibonacci

class FibonacciTest(unittest.TestCase):
    def test_value_0_should_return_0(self):
        fibonacci = Fibonacci()
        self.assertEqual(0, fibonacci.calculate(0))

    def test_value_1_should_return_1(self):
        fibonacci = Fibonacci()
        self.assertEqual(1, fibonacci.calculate(1))

    def test_value_10_should_return_55(self):
        fibonacci = Fibonacci()
        self.assertEqual(55, fibonacci.calculate(10))

    def test_value_minus1_should_return_exception(self):
        fibonacci = Fibonacci()
        self.assertRaises(ValueError, fibonacci.calculate, -1)

    def test_value_93_should_return_exception(self):
        fibonacci = Fibonacci()
        self.assertRaises(ValueError, fibonacci.calculate, 93)
```

Test execution order is not secure, do no write test methods (or classes) that depend on others!!!

Assertions on real numbers

When dealing with real numbers, an “exact” assertion may be not feasible

- E.g., due to number approximations

unittest provides a useful assertion:

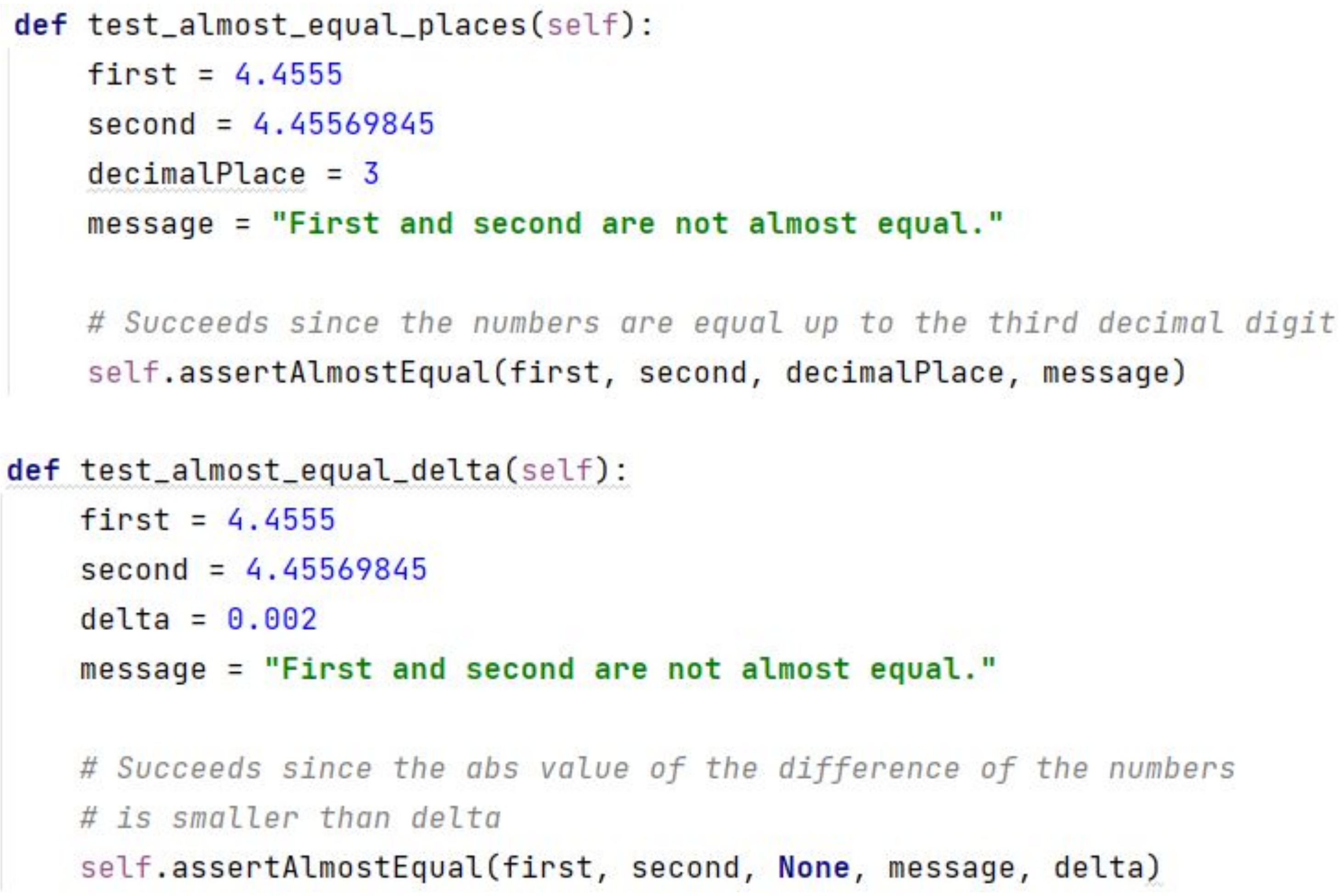
```
assertAlmostEqual(first, second, places=7, msg=None, delta=None)
```

It asserts that two real numbers (i.e., expected and actual) are equal within a positive delta

Assertions on real numbers

```
assertAlmostEqual(first, second, places=7, msg=None, delta=None)

first: first input number

second: second input number

places: how many decimal places are considered for approximation

delta: delta value for approximation
```

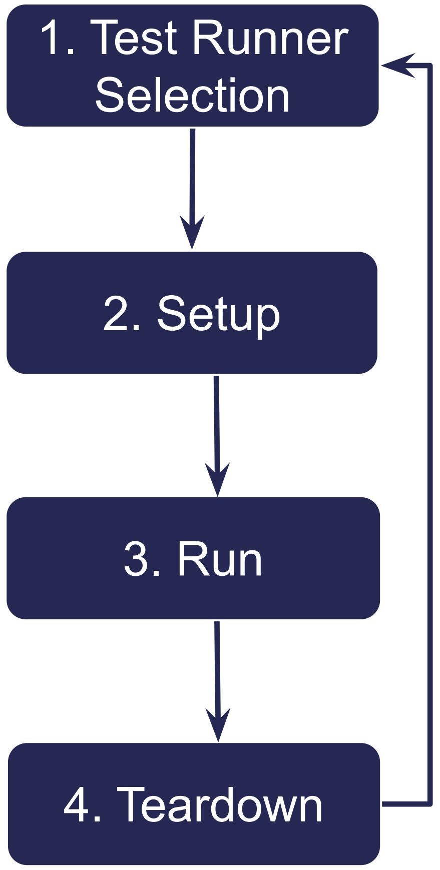
Assertions on real numbers



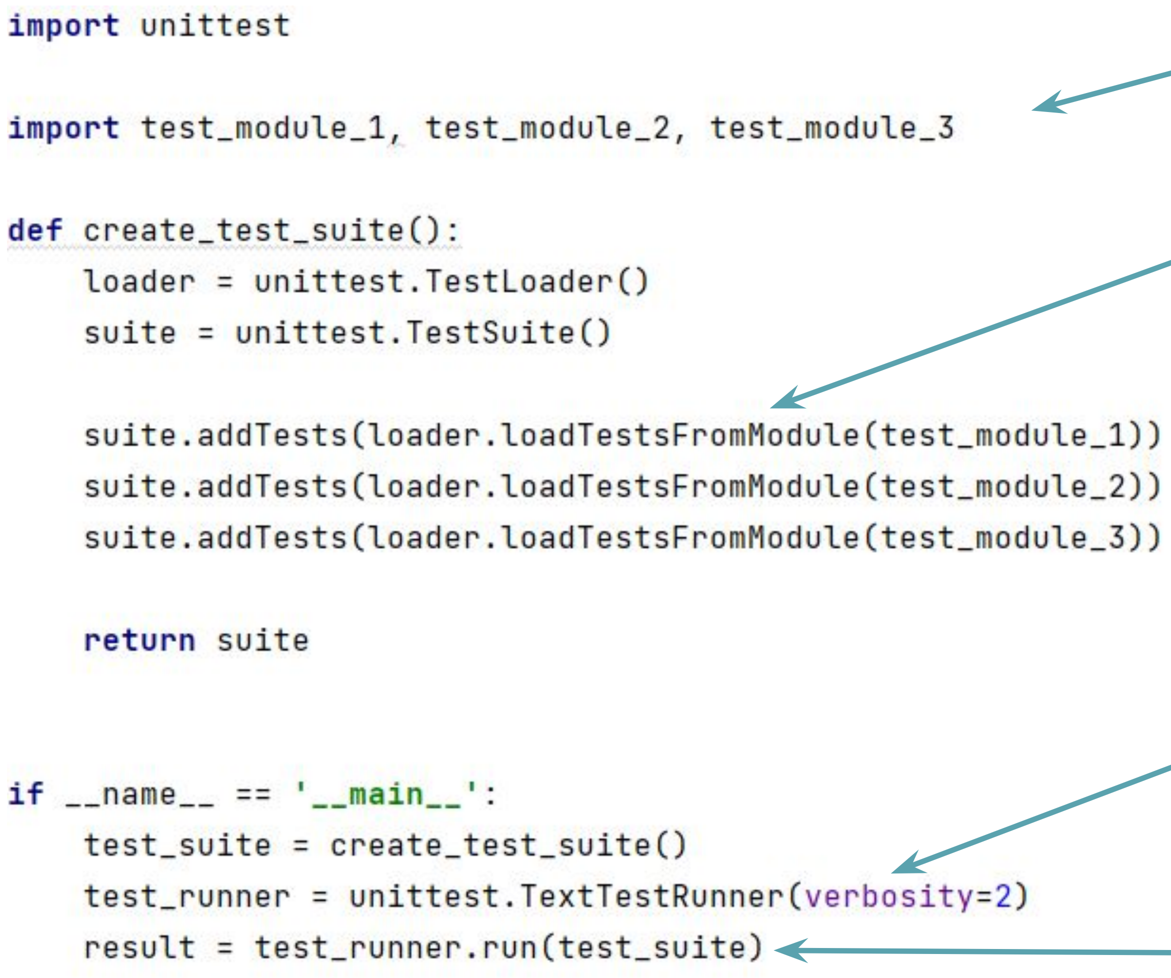
```python
def test_almost_equal_places(self):
    first = 4.4555
    second = 4.45569845
    decimalPlace = 3
    message = "First and second are not almost equal."

    # Succeeds since the numbers are equal up to the third decimal digit
    self.assertAlmostEqual(first, second, decimalPlace, message)

def test_almost_equal_delta(self):
    first = 4.4555
    second = 4.45569845
    delta = 0.002
    message = "First and second are not almost equal."

    # Succeeds since the abs value of the difference of the numbers
    # is smaller than delta
    self.assertAlmostEqual(first, second, None, message, delta)
```

unittest execution cycle

1. Select a test runner, which creates a new instance of the test class
2. Invoke the setUp method on the test class (if any)
3. Run a test method
4. Invoke the tearDown method on the test class (if any)



Execute the entire test suite



```python
import unittest

import test_module_1, test_module_2, test_module_3

def create_test_suite():
    loader = unittest.TestLoader()
    suite = unittest.TestSuite()

    suite.addTests(loader.loadTestsFromModule(test_module_1))
    suite.addTests(loader.loadTestsFromModule(test_module_2))
    suite.addTests(loader.loadTestsFromModule(test_module_3))

    return suite


if __name__ == '__main__':
    test_suite = create_test_suite()
    test_runner = unittest.TextTestRunner(verbosity=2)
    result = test_runner.run(test_suite)
```

Test classes

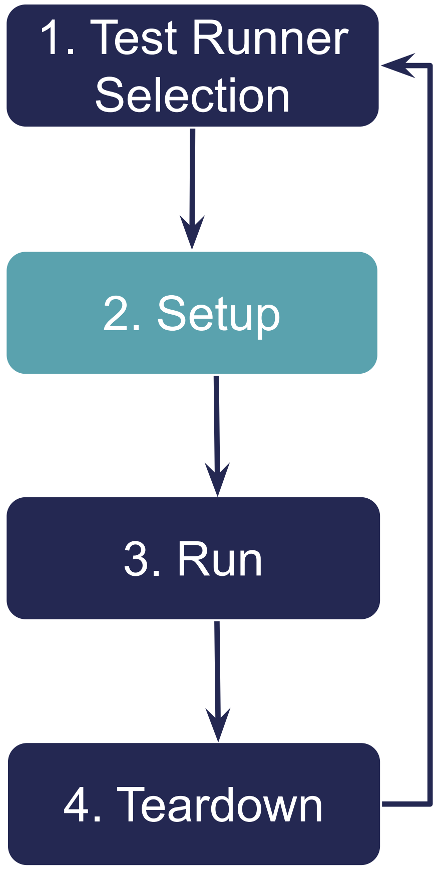
Build the test suite by loading the test cases from each test module

How detailed the output should be

Execute the test suite

unittest execution cycle

1. Select a test runner, which creates a new instance of the test class
2. <u>Invoke the setup method on the test class (if any)</u>
3. Run a test method
4. Invoke the teardown method on the test class (if any)



Setup method

Called before each test case. It helps creating a **test fixture**, namely:

A set of objects needed to consistently run the tests

It is defined as follows (method names must be exactly as below):

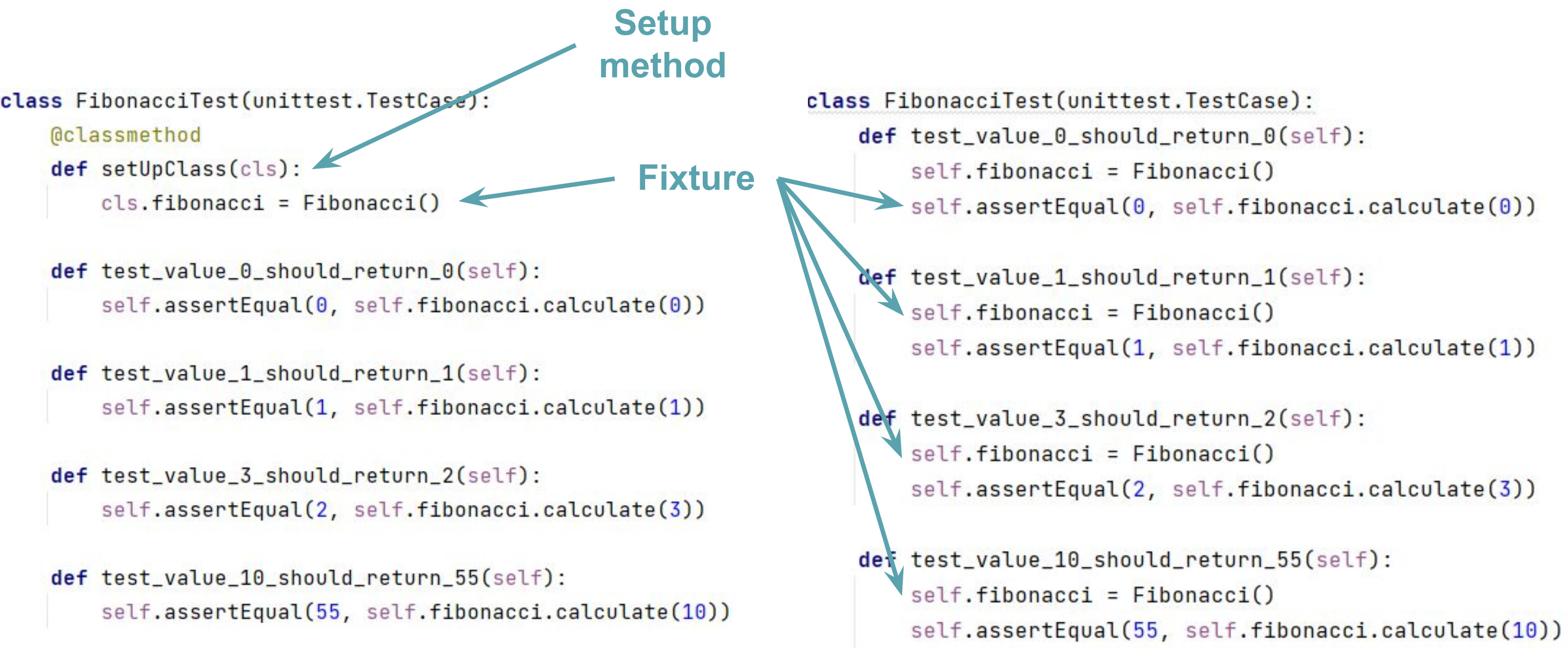
```
@classmethod
def setUpClass(cls): ...
```

It is run once, before all test cases

```
def setUp(self): ...
```

It is run before each test method

Setup method example

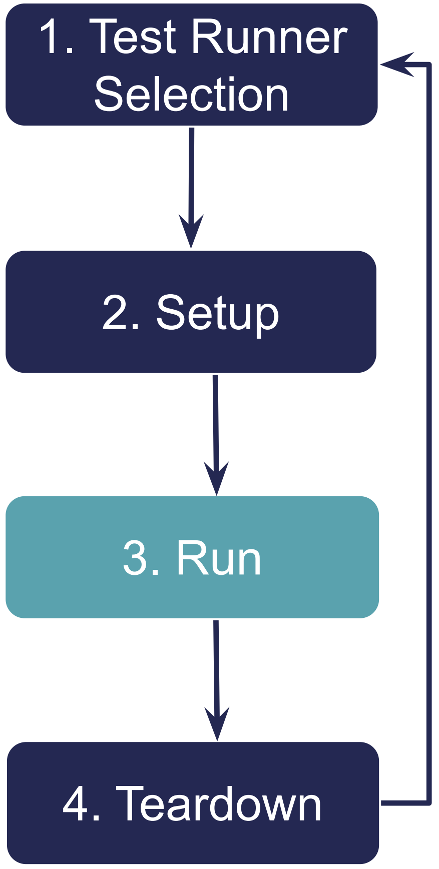


```
class FibonacciTest(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls.fibonacci = Fibonacci()

    def test_value_0_should_return_0(self):
        self.assertEqual(0, self.fibonacci.calculate(0))

    def test_value_1_should_return_1(self):
        self.assertEqual(1, self.fibonacci.calculate(1))

    def test_value_3_should_return_2(self):
        self.assertEqual(2, self.fibonacci.calculate(3))

    def test_value_10_should_return_55(self):
        self.assertEqual(55, self.fibonacci.calculate(10))
```

```
class FibonacciTest(unittest.TestCase):
    def test_value_0_should_return_0(self):
        self.fibonacci = Fibonacci()
        self.assertEqual(0, self.fibonacci.calculate(0))

    def test_value_1_should_return_1(self):
        self.fibonacci = Fibonacci()
        self.assertEqual(1, self.fibonacci.calculate(1))

    def test_value_3_should_return_2(self):
        self.fibonacci = Fibonacci()
        self.assertEqual(2, self.fibonacci.calculate(3))

    def test_value_10_should_return_55(self):
        self.fibonacci = Fibonacci()
        self.assertEqual(55, self.fibonacci.calculate(10))
```

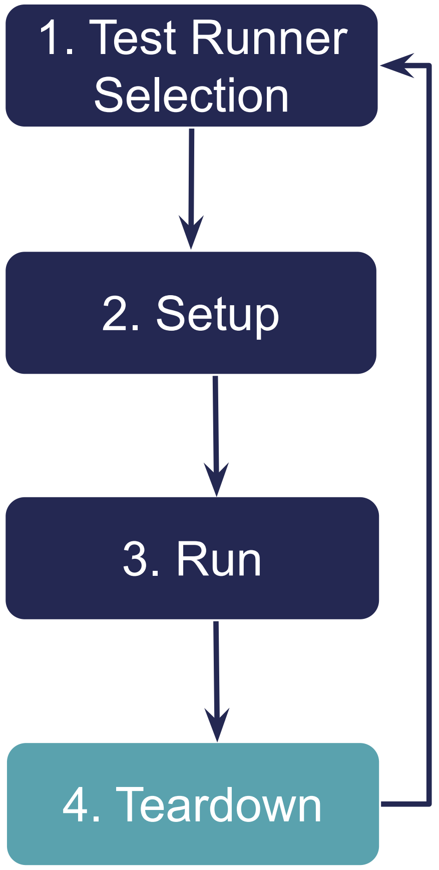
unittest execution cycle

1. Select a test runner, which creates a new instance of the test class
2. Invoke the setup method on the test class (if any)
3. <u>Run a test method</u>
4. Invoke the teardown method on the test class (if any)



unittest execution cycle

1. Select a test runner, which creates a new instance of the test class
2. Invoke the setup method on the test class (if any)
3. Run a test method
4. Invoke the teardown method on the test class (if any)



Teardown method

Called after each test case execution. It is directed to restore the environment to the same condition it was before the test execution.

It is defined as follows (method names must be exactly as below) :

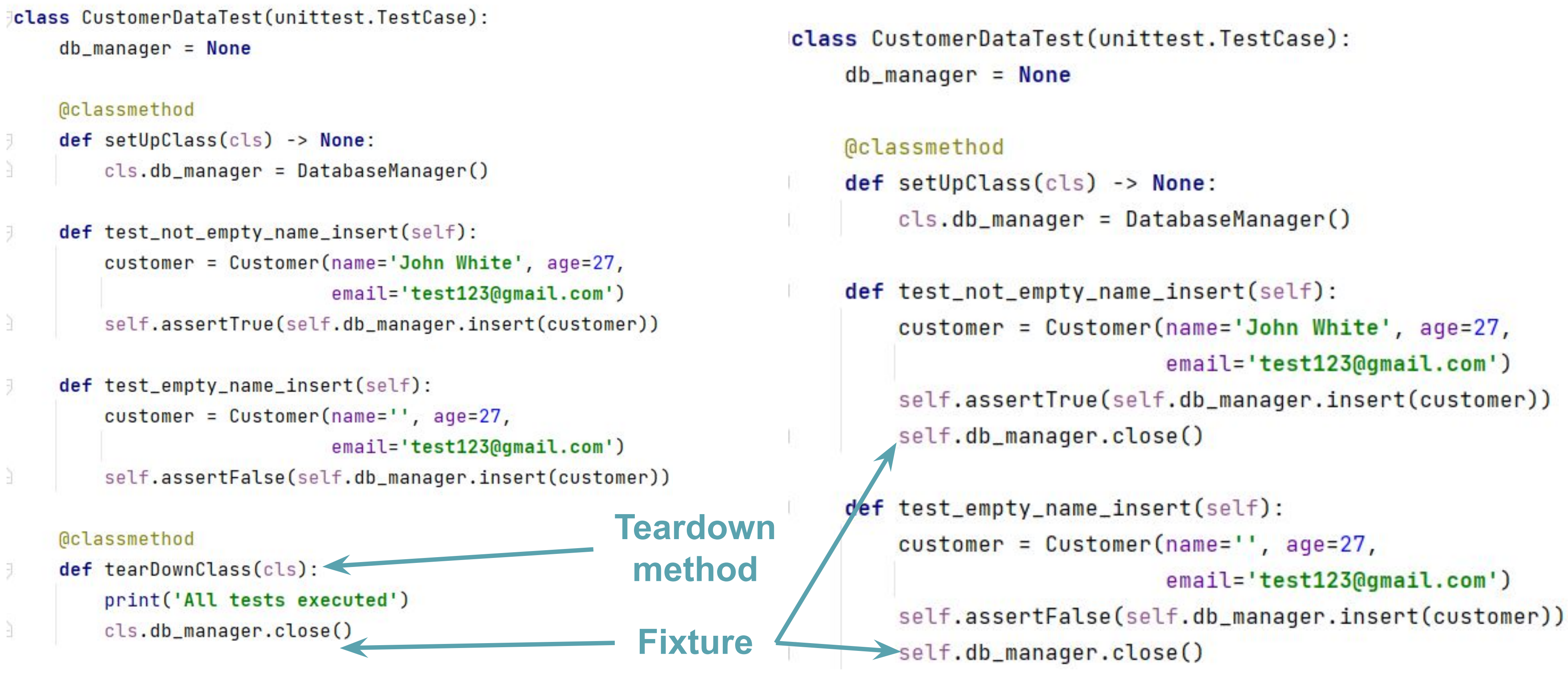
```
@classmethod
def tearDownClass(cls): ...
```

It is run after each test method

```
def tearDown(self): ...
```

It is run once after all tests have been executed

Teardown method example



```
class CustomerDataTest(unittest.TestCase):
    db_manager = None

    @classmethod
    def setUpClass(cls) -> None:
        cls.db_manager = DatabaseManager()

    def test_not_empty_name_insert(self):
        customer = Customer(name='John White', age=27,
                            email='test123@gmail.com')
        self.assertTrue(self.db_manager.insert(customer))

    def test_empty_name_insert(self):
        customer = Customer(name='', age=27,
                            email='test123@gmail.com')
        self.assertFalse(self.db_manager.insert(customer))

    @classmethod
    def tearDownClass(cls):
        print('All tests executed')
        cls.db_manager.close()
```

```
class CustomerDataTest(unittest.TestCase):
    db_manager = None

    @classmethod
    def setUpClass(cls) -> None:
        cls.db_manager = DatabaseManager()

    def test_not_empty_name_insert(self):
        customer = Customer(name='John White', age=27,
                            email='test123@gmail.com')
        self.assertTrue(self.db_manager.insert(customer))
        self.db_manager.close()

    def test_empty_name_insert(self):
        customer = Customer(name='', age=27,
                            email='test123@gmail.com')
        self.assertFalse(self.db_manager.insert(customer))
        self.db_manager.close()
```