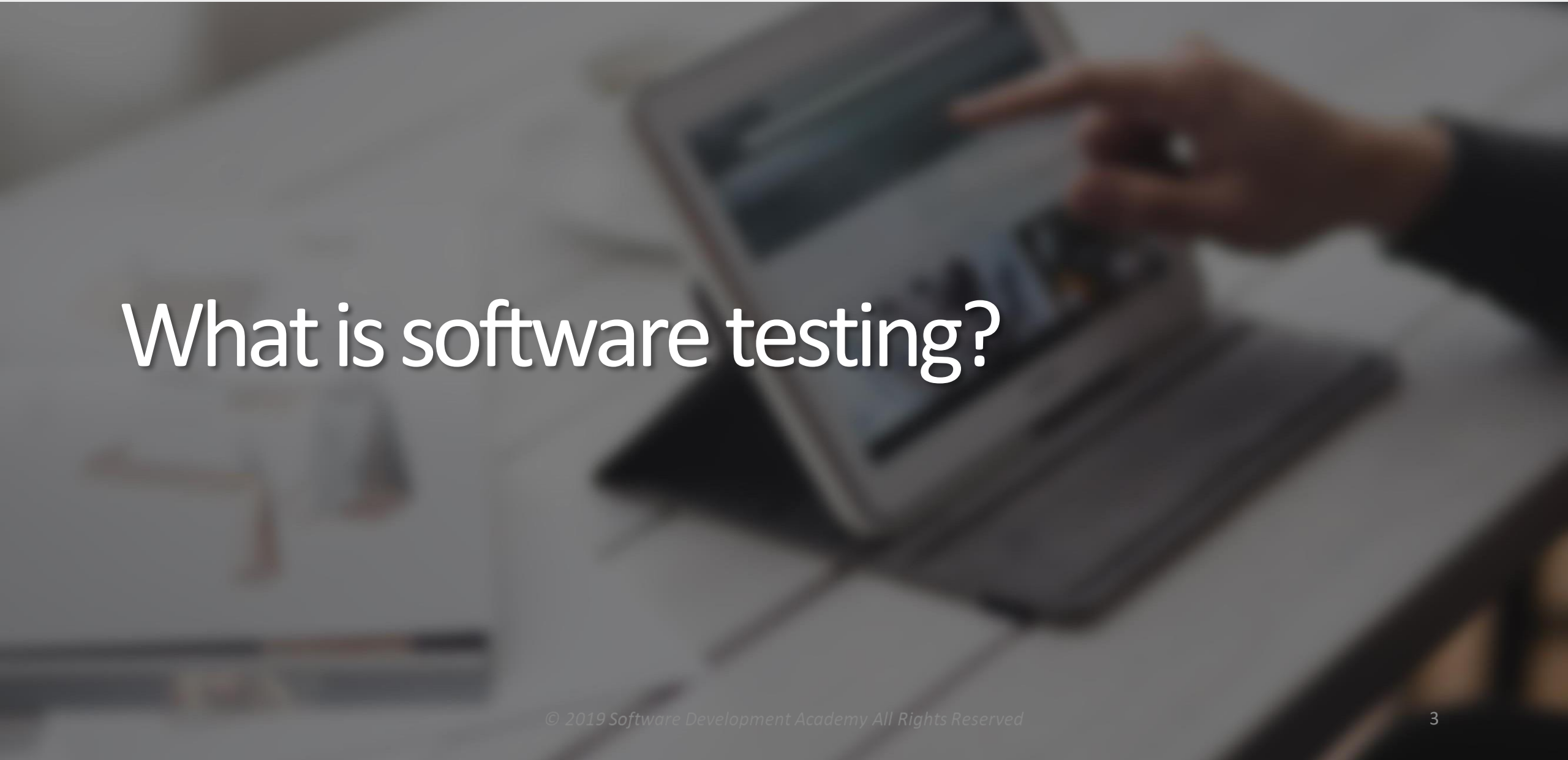
Software Testing Fundamentals

Agenda

1. What is software testing?
2. unittest - built-in test framework
3. Organizing test
4. Test suites
5. Test-driven Development
6. Library – testing exercise



What is software testing?

What is software testing?

- Certain parts of the code and the whole application needs to work as we intend it to
- Software testing is creating and running tests which check whether that's true
- The tests are not a part of the application itself

Why is it important?

- Small applications are easy to test it just by running them
- Larger codebases it gets increasingly difficult to test all the functions and their outcomes
- It is a good practice to test your code every time you introduce a new feature
- This is why automated software testing is so important

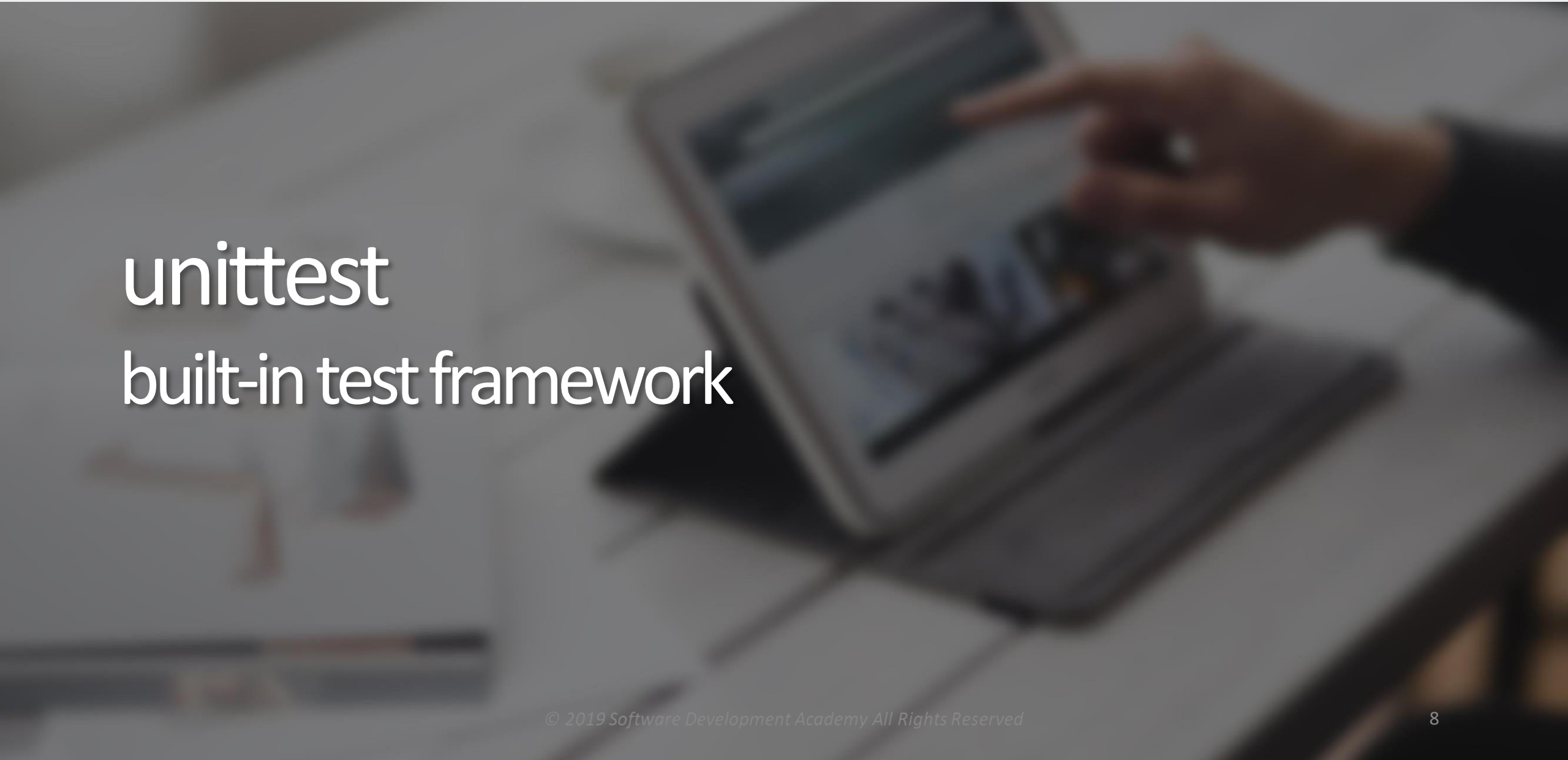
Testing methods

There are many different kinds of tests, just to name a few:

- **Unit tests** - our focus for this module
- Integration tests
- Performance tests
- Security tests
- System tests
- And many more

Unit tests

- Unit tests are the most basic, yet powerful, type of tests
- They should test only a small chunk of code - a function or even a part of it
- A function or a section of code can be tested by multiple tests
- It is common to try and test all the code branches and corner cases



unittest

built-in test framework

unittest – built-in test framework

- unittest is Python's built-in test framework
- As its name suggests, it is meant for creating unit tests
- Let's see an example of unittest in action

Example 01-unittest/example-01.py

```python
import unittest


class TestBuiltins(unittest.TestCase):
    def test_membership(self):
        self.assertIn("A", "Andalusia")
        self.assertTrue("A" in "Andalusia")

    def test_instances(self):
        self.assertIsInstance(5, int)
        self.assertTrue(isinstance(5, int))

    def test_falsehood(self):
        self.assertFalse(False)


if __name__ == "__main__":
    unittest.main()
```

```
$ python3 01-unittest/example-01.py
```

```
...
----------------------------------------
Ran 3 tests in 0.001s

OK
```

Example explanation

1. We have create a new test case by creating `class TestBuiltins` which inherits `unittest.TestCase`. (Inheritance will be explained in further modules)
2. The test case contains three separate tests - `def test_*` functions.
3. Test function names have to start with `test`
4. Each of the functions invokes at least one `self.assert*` function
5. The test case and its functions are discovered by `unittest` when the script is started

Exercise 1

1. Try changing one of the `test_*` functions, so that the statements in it are **not** true.
2. Re-run the tests.
3. What happened?

unittest discovery

Unittest can be used to call individual test modules/cases/functions like so:

```
$ python3 -m unittest <file path or import path>
```

Example 01-unittest/example-02.py

```
$ python3 -m unittest 01-unittest/example-02.py
OR
$ python3 -m unittest 01-unittest.example-02
OR
$ python3 -m unittest 01-unittest.example-02.TestBuiltins
```

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s
OK
```

```
$ python3 -m unittest 01-unittest.example-02.TestBuiltins.test_instances
```

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s
OK
```

Exercise 2

1. Run only test_slicing function from `01-unittest/exercise-02.py` by using `python3 -m unittest <import path>`

Example 01-unittest/example-03.py

- Unittest can automatically discover tests inside a folder
- Run discovery using `python3 -m unittest discover <directory>`
- It will look for all files named `test*.py` and run tests inside them
- You can change filename pattern using `-p <new pattern>` option

```
$ python3 -m unittest discover 01-unittest/example-03
```

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s
OK
```

Exercise 3

1. Change default pattern in `python3 -m unittest discover 01-unittest/example-03 -p test*.py` call to run all the tests in `example-03` directory.

unittest flow control

1. Sometimes we expect more from our tests:
   - a test needs to be skipped for some reason (it is platform specific, it is not ready)
   - a test needs some setup that's not part of the test (database connection, file creation)
   - a test needs some cleanup after it runs (closing db connection, removal of temporary files)
   - we expect a certain test to fail with an error or exception

2. This is where the following decorators and methods (both types called fixtures) come handy:
   - `unittest.skip, unittest.skipIf, unittest.skipUnless`
   - `setUp`
   - `tearDown`
   - `unittest.expectedFailure`

Example 01-unittest/example-04.py

```python
import unittest
import tempfile
import time
import sys

class TestOnTemporaryFile(unittest.TestCase):
    def setUp(self):
        print(f"Running {self.__class__} setup")
        self.tmp_file = tempfile.TemporaryFile(mode="w+t")

    def tearDown(self):
        print(f"Running {self.__class__} teardown")
        self.tmp_file.close()

    def test_file_write(self):
        print("Running test_file_write test")
        bytes_written = self.tmp_file.write("Hello!")
        self.assertEqual(bytes_written, 6)

class TestPlatformSpecific(unittest.TestCase):
    @unittest.skip("This test is not ready")
    def test_not_ready(self):
        self.fail("I should have been skipped!")

    @unittest.skipUnless(sys.platform == "linux", "requires linux")
    def test_get_boottime_clock(self):
        self.assertIn("CLOCK_BOOTTIME", dir(time))

    @unittest.skipIf(sys.version_info.major < 3, "For Python 3.X only")
    def test_string_instance(self):
        self.assertEqual(5 / 2, 2.5)  # this fails for Python 2.X
        self.assertEqual(5 // 2, 2)
```

```
$ python3 -m unittest 01-unittest/example-04.py
```

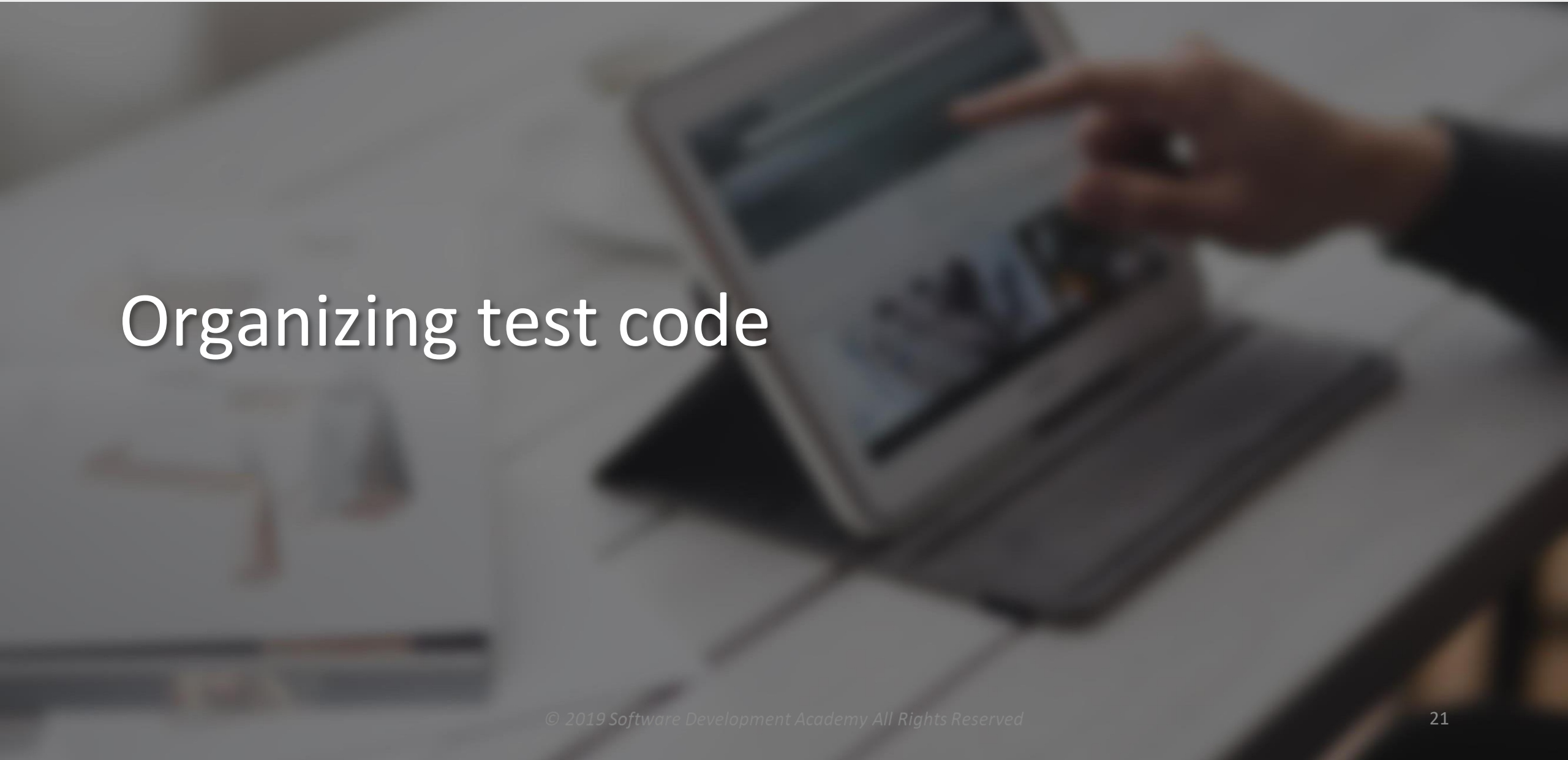
```
Running <class '01-unittest.example-04.TestOnTemporaryFile'> setup
Running test_file_write test
Running <class '01-unittest.example-04.TestOnTemporaryFile'> teardown
..s.
----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK (skipped=1)
```

Exercise 01-unittest/exercise-03.py

You have been given a `DatabaseInterface` class and a test case for it, however there are several things for you to fix:

1. Each of the tests takes a lot of time, because of `DatabaseInterface.__init__`. Fix that without removing the sleep call. Hint: maybe it is enough to clear the database before each test?
2. A test for `get_record` is missing. Add it.
3. One of the tested methods is not implemented yet. Skip the test related to it using `unittest.skip()`, or handle the error using `self.assertRaises()`



Organizing test code

Test case recap

1. Test cases are basic building block of unit testing
2. More than one test can be performed during a test case
3. `test_*` functions are launched alphabetically
4. `setUp` and `tearDown` are run at the beginning and at the end of the test case respectively

Test suites

It is recommended to group test cases together, according to features they test. We can use `TestSuite` for this.

Example:

1. `EshopUserPanel` test suite
   - login test case
   - logout test case
   - admin login test case
   - admin logout test case
2. `EShopCart` test suite
   - add item test case
   - remove item test case
   - clear cart test case

Test suites

Example 02-test-suites/example-01.py

```python
import unittest
from game import Enemy
from game import Player


class EnemyInitTest(unittest.TestCase):
    def test_enemy_init(self):
        e = Enemy("orc")
        self.assertEqual(e.name, "orc")
        self.assertEqual(e.lives, 1)


class EnemyDamageTest(unittest.TestCase):
    def test_enemy_takes_damage(self):
        e = Enemy("goblin")
        e.receive_damage(1)
        self.assertEqual(e.lives, 0)


class PlayerMovement(unittest.TestCase):
    def setUp(self):
        self.player = Player("Link")

    def test_move_north(self):
        self.player.reset_position()
        self.player.move_north()
        self.assertEqual(self.player.position_xy, (0, 1))

    def test_move_east(self):
        self.player.reset_position()
        self.player.move_east()
        self.assertEqual(self.player.position_xy, (1, 0))
```

```python
    def test_pretty_position(self):
        self.player.move_south()
        self.player.move_south()
        self.player.move_west()
        self.player.move_west()
        self.assertEqual(
            self.player.pretty_position(),
            "Link went 2 steps south and 2 steps west"
        )


enemySuite = unittest.TestSuite()
enemySuite.addTests(
    [
            EnemyInitTest("test_enemy_init"),
            EnemyDamageTest("test_enemy_takes_damage"),
    ]
)
```

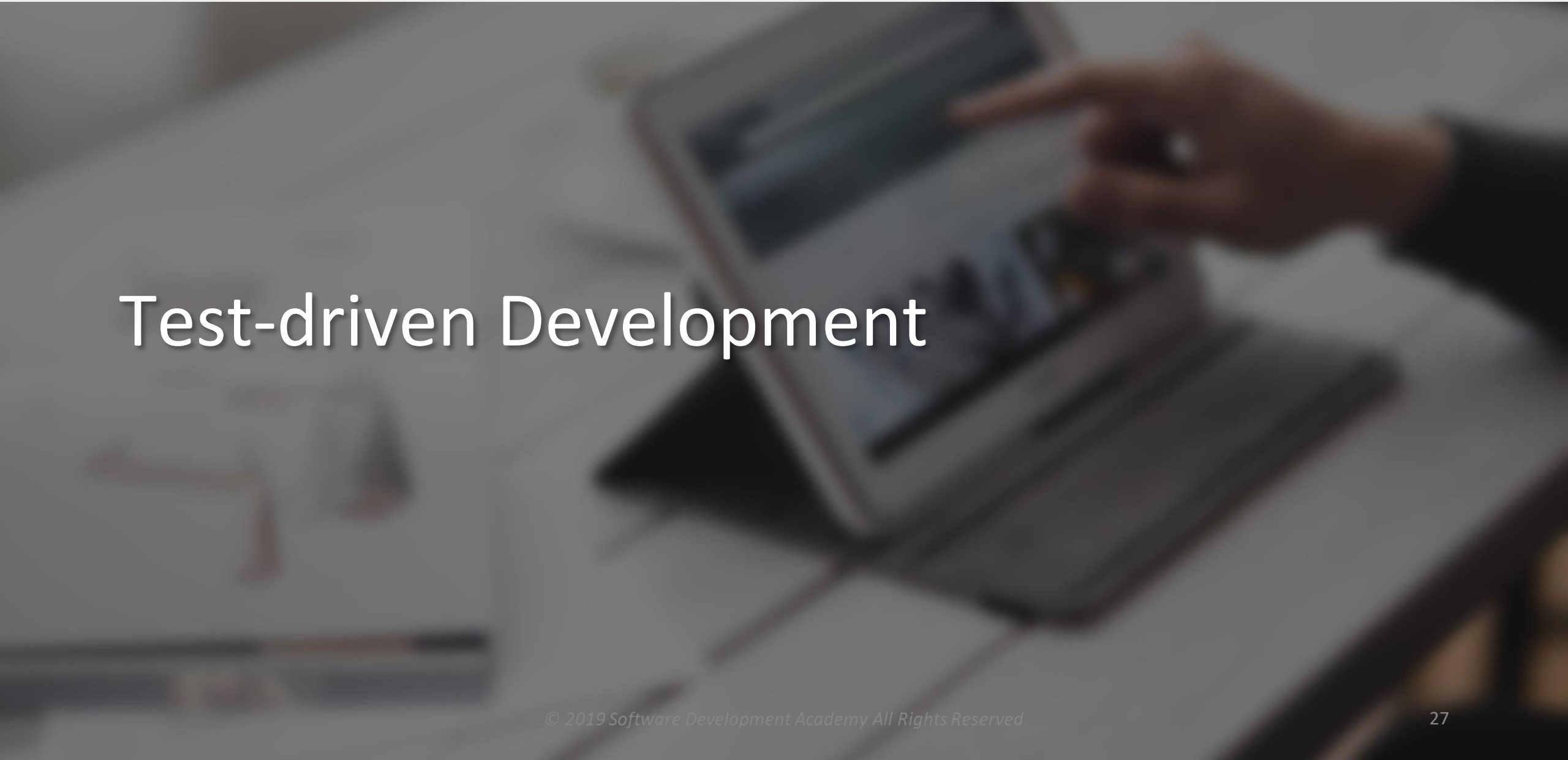
```
$ python3 -m unittest 02-test-suites.example-01.enemySuite
```

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Exercise – test suites

1. Add a test suite containing player movement tests.
2. Add a new test case called `PlayerDamage` with the following functions:
   - `setUp` which creates a single `Player` instance and a single `Enemy` instance
   - `test_player_attack` in which the `self.player.attack_enemy(self.enemy)` is tested
3. Add `PlayerDamage` to both `playerSuite` and `enemySuite`.



Test-driven Development

Test-driven Development

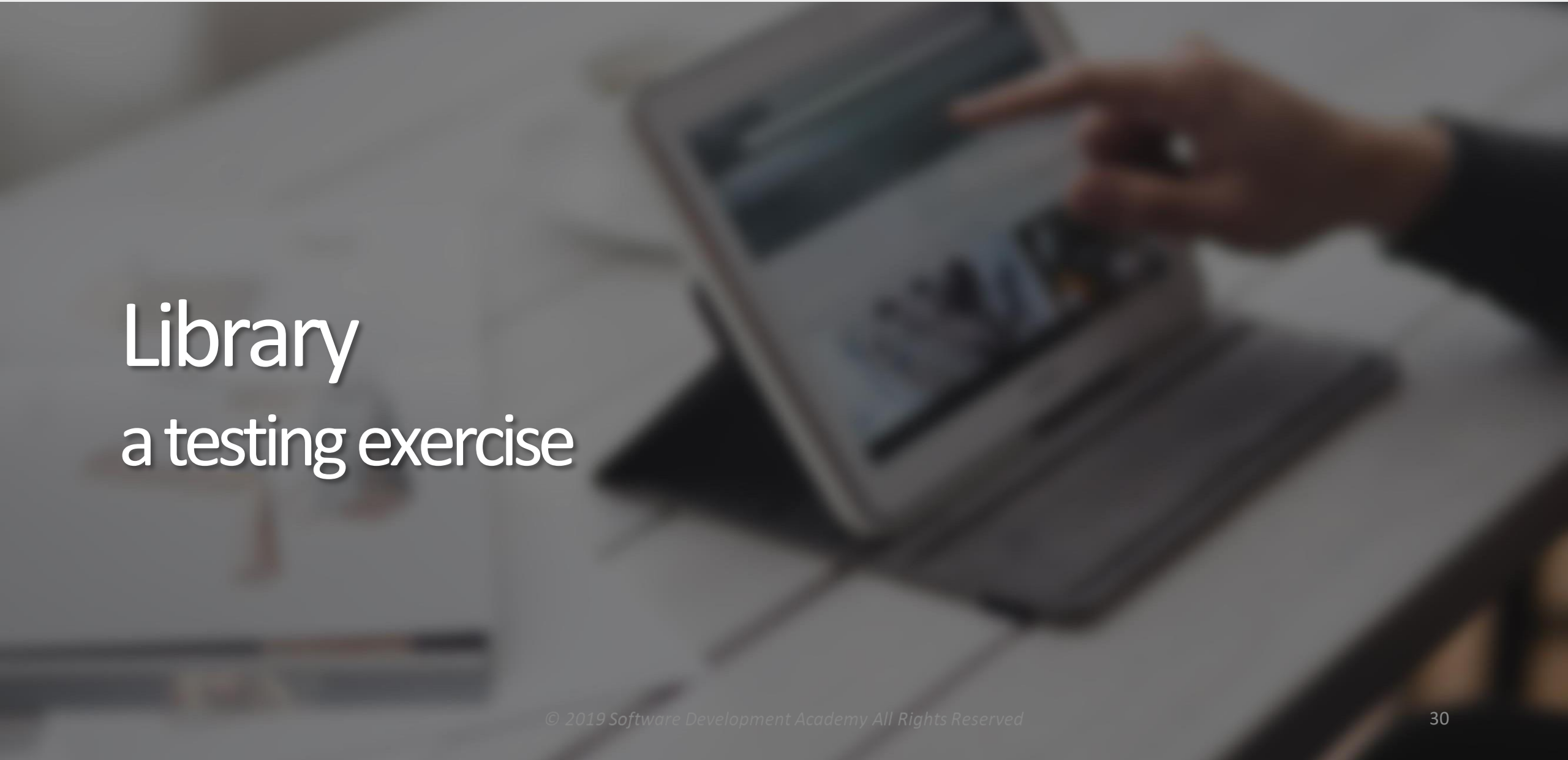
Test-driven development is a methodology often used in the industry. It can be captured in the following steps:

1. Add a new test for a functionality that is not yet developed
2. Run tests to make sure the new test fails
3. Implement the new functionality not minding the code quality too much
4. Run tests to make sure the new test passes
5. Refactor = Improve quality of code used in implementation

Exercise

1. Write a test for `Tasks.today()` , which is not yet implemented. You will find a description of expected behaviour in the docstring. *
2. Run tests to ensure they fail
3. Implement `today()`**
4. Run tests to ensure they succeed
5. Refactor code if needed

*, ** If you are unsure how to proceed you can check out files in solution subdirectory



Library

a testing exercise

Exercise

1. Write tests cases and suites for `Book` and `Library` classes in `04-library-exercise/library.py`
2. Take a look at example solution provided in `04-library-exercise/solution.py` after you are finished
3. What are the differences between your approach and the approach presented?
4. Try writing a few cases similar to the ones in given solution