UNIT TESTING IN PYTHON USING **PyTest**

MODULE 3

SOFTWARE QUALITY AND TESTING

Software Quality

- Software is the collection of computer programs, related data, and associated documentation developed for a particular customer or for a general market.
- The question *What is software quality?* can generate different answers, depending on the involved practitioner's role in a software system.
- There are two main groups of people involved in a software product or service:
  - **Consumers**
  - **Producers**

Software Quality

- The quality expectations of consumers are that a software system performs ***useful functions*** as specified.
- For software producers, the fundamental quality question is fulfilling their contractual obligations by producing software products that conform to the **Service Level Agreement** (**SLA**).
- The definition of software quality by the well-known software engineer Roger Pressman comprises both points of view:

  *An effective software process applied in a manner that creates a useful product that provides **measurable value** for those who produce it and those who use it.*

Quality engineering

> Quality engineering is a process that evaluates, assesses, and improves the quality of software.

> There are three major groups of activities in the quality engineering process:

1. Quality planning
2. Quality Assurance (QA)
3. Post-QA

Quality engineering

> These phases are represented in the following chart:

Requirements and specification

- A requirement is a statement identifying a capability, physical characteristic, or quality factor that bounds a product or process need for which a solution will be pursued.
- There are two kinds of software requirements
  - **Functional requirements**
  - **Non-functional requirements**

Quality Assurance

- **Quality Assurance** (**QA**) is primarily concerned with defining or selecting standards that should be applied to the software development process or software product.
- Daniel Galin, the author of the book *Software Quality Assurance* (2004) defined QA as:

  Systematic, planned set of actions necessary to provide adequate confidence that the software development and maintenance process of a software system product conforms to established specification as well as with the managerial requirements of keeping the schedule and operating within the budgetary confines.

ISO/IEC-2500*n*

> **ISO/IEC-2500** series is an international standards on **Software product Quality Requirements and Evaluation** (**SQuaRE**)

> The ISO/IEC-2500 quality reference model distinguishes different views on software quality:

- **Internal quality**
- **External quality**
- **Quality in use**

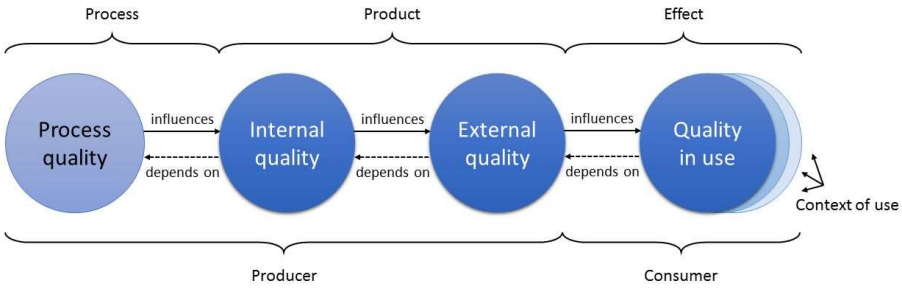
ISO/IEC-2500*n*

> Ideally, the development (*process quality*) influences the internal quality

> Then, the internal quality determines the external quality.

> Finally, external quality determines quality in use.

> This chain is depicted in the following picture:



ISO/IEC-2500*n*

- The quality model of ISO/IEC-2500 divides the product quality model into eight top-level quality features
  - Functional suitability
  - Performance efficiency
  - Compatibility
  - Usability
  - Reliability
  - Security
  - Maintainability
  - Portability

Verification and Validation

- Verification and Validation is concerned with evaluating that the software being developed meets its specifications and delivers the functionality expected by the consumers.
- The difference between them:
  - **Verification**: *are we building the product right?*
  - **Validation**: *are we building the right product?*

Verification and Validation

> V&V activities include a wide array of QA activities.

> Within the V&V process, two big groups of techniques of system checking and analysis may be used:

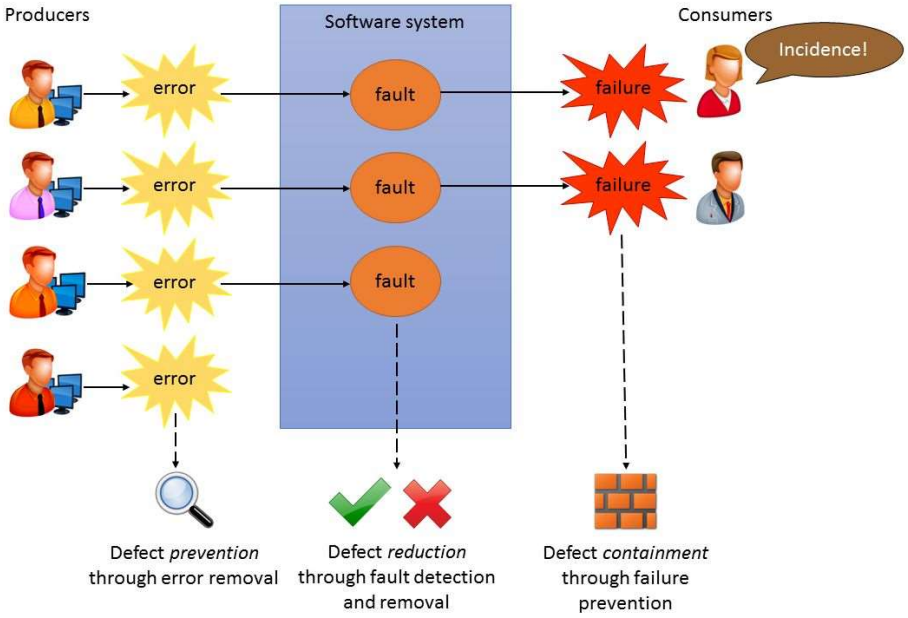
- **Software testing**
- **Static analysis**

Software defects

> The term **defect** refers to a generic software problem. The IEEE Standard 610.12 propose the following taxonomy related to software defects:

- **Error**
  - Syntax error
  - Logic error
- **Fault**
- **Failure**

Software Defect Chain & associated QA Activities



Static Analysis

> There are several advantages to software analysis over testing:

1. During testing, errors can hide other errors
2. Incomplete versions of a system can be statically analyzed without additional cost
3. Static analysis can consider broader quality attributes of a software system, such as compliance with standards, portability, and maintainability.

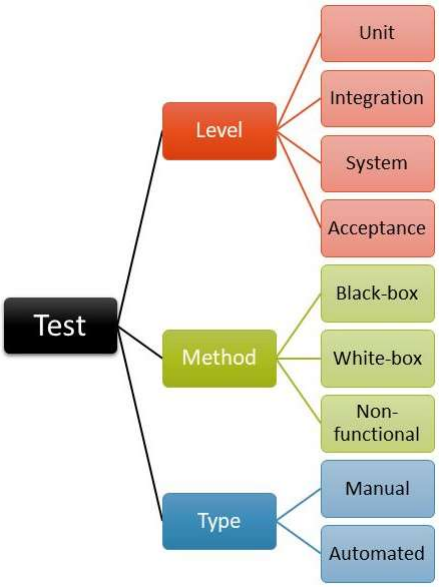
Static Analysis

- There are different methods that can be identified as static analysis:
  - **Inspection**
  - **Review**
  - **Automated software analysis**

Software Testing

- Software testing consists of the dynamic evaluation of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior.
- The key concepts of this definition
  - **Dynamic**
  - **Finite**
  - **Selected**
  - **Expected**

Taxonomy of Software Testing



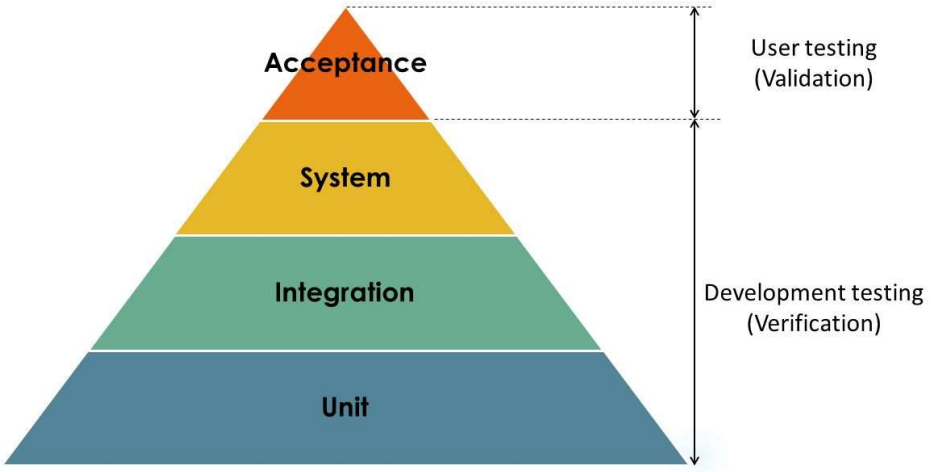
Testing Levels

> Depending on the size of the SUT and the scenario in which it is exercised, testing can be carried out at different levels.

> We classify the different testing levels in four phases:

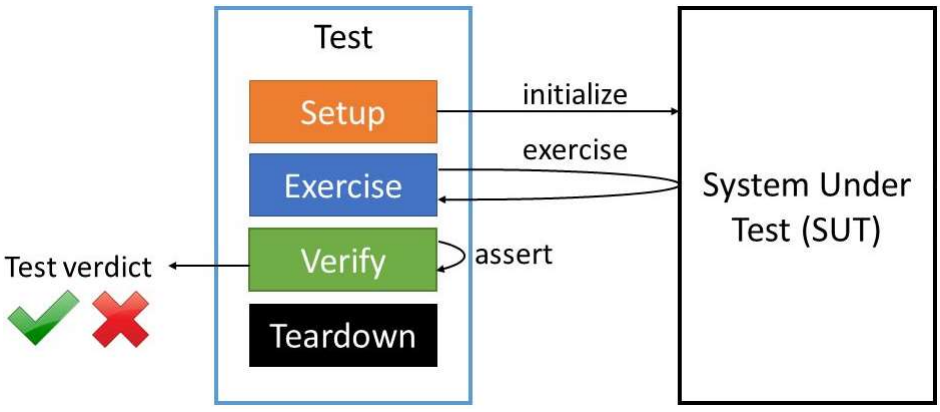
- **Unit testing**
- **Integration testing**
- **System testing**
- **Acceptance testing**

Testing Levels



Unit Tests Structure

> In general, a unit test is composed by four stages



Test Doubles

> Unit testing is done with the unit under test in isolation, that is, without interacting its DOCs. To that aim, *test doubles* are employed to replace any components on which the SUT depends:
  - A **dummy** object simply satisfies the real object API but it is never actually used
  - A **fake** object replaces the real object with a simpler implementation, for example, an in-memory database.
  - A **stub** object replaces the real object providing hard-coded values as responses.
  - A **mock** object also replaces the real object, but this time with programmed expectations as responses.
  - A **spy** object is a partial mock object

Testing Methods

> Testing methods (or strategies) define the way for designing test cases:
  - Responsibility based (black-box)
  - Implementation based (white box)
  - Nonfunctional
> Black-box techniques design test cases on the basis of the specified functionality of the item to be tested.
> White-box ones rely on source code analysis to develop test cases.
> Hybrid techniques (grey-box) testing designs test cases using both responsibility-based and implementation-based approaches.

Black-box testing

- **Black-box testing** (also known as **functional** or **behavioral** testing) is based on requirements with no knowledge of the internal program structure or data.
- The most well-known black-box testing are
  - **Systematic testing**
  - **Random testing**
  - **Graphic User Interface** (**GUI**) **testing**
  - **Model-based testing** (**MBT**)
  - **Smoke testing**
  - **Sanity testing**

White-box Testing

- **White-box Testing/Structural Testing**
  - **Code coverage**
    1. Statement coverage
    2. Decision (branch) coverage
    3. Condition coverage
    4. Paths coverage
    5. Function coverage
    6. Entry/exit coverage
  - **Fault injection**
  - **Mutation testing**

Non-functional Testing

- The **non-functional** aspects of a system can require considerable effort to test.
- Performance testing is to evaluate the compliance of a SUT with specified performance requirements.
- These requirements usually include constraints about the time behavior and resource usage.
  - *Performance testing*
  - *Load testing*
  - *Volume testing*
  - *Stress testing*
  - *Security testing*
  - *Usability testing*
  - *Accessibility testing*

Testing Types

- There are two main types to carrying out software testing:
  - **Manual testing**
  - **Automated testing**

Other Testing Approaches

- *User or customer testing* is a stage in the testing process in which users or customers provide input and advice for system testing.
  - *Acceptance testing* is a type of user testing, but there can also be different types of *user testing*:
    - **Alpha testing**
    - **Beta testing**
    - **Operational testing**
  - R*elease testing*
    - The primary goal of the release testing process is to convince the supplier of the system that is good enough for use.

Principals

- Minimize Untestable Code
- No Test Logic in Production Code
- Verify One Condition per Test
- Test Concerns Separately
- Keep Tests Independent
- Use the Front Door First (do not have 'special' test only ways of using a class)
- Communicate Intent (tests are great design/ usage 'docs')
- Easy to Setup
- FAST!

Minimize Untestable Code

> Some kinds of code are difficult to test using Fully Automated Tests
  - GUI components
  - Multi-threaded code
> The problem all these kinds of code share is being embedded in a context that makes it hard to instantiate or interact with them from automated tests.
> Makes it harder to refactor safely and more dangerous to modify to introduce new functionality.

No Test Logic in Production Code

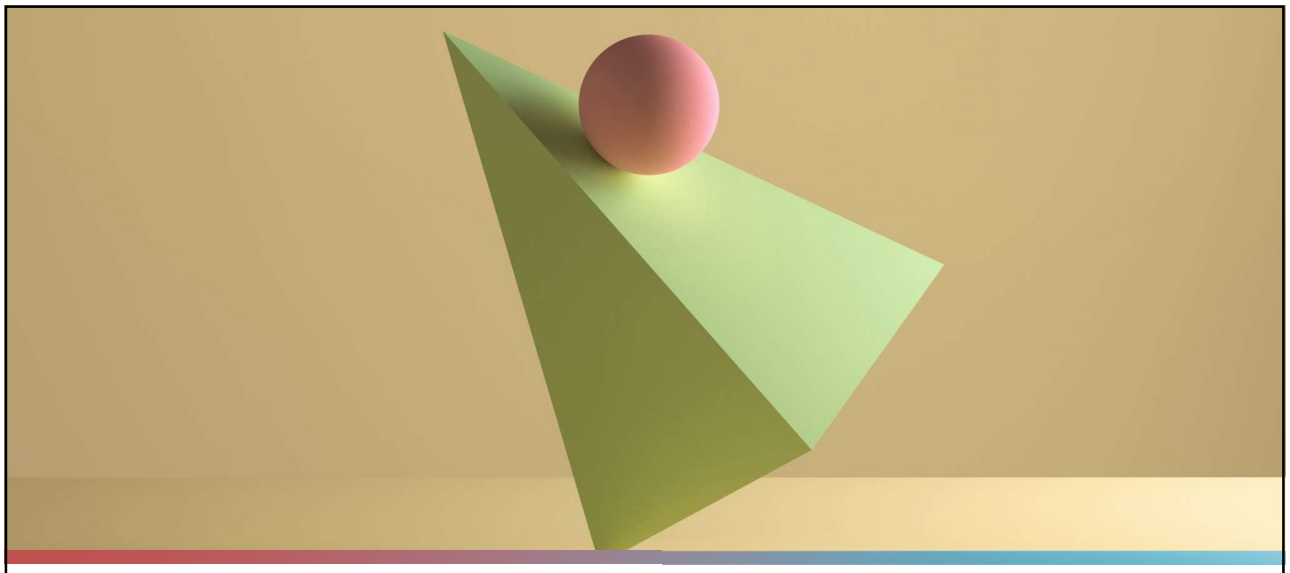
> When the production code hasn't been designed for testability, we may be tempted to put "hooks" into the production code to make it easier to test.
> These hooks typically take the form of if testing then ... and may either run alternate logic or may prevent certain logic from running.
> Testing is about verifying the behavior of a system.
  - If the system behaves differently when under test then how can we be certain that the production code actually works?
  - Even worse, the test hooks could cause the software to fail in production!

Verify One Condition per Test

- Many tests require a starting state other than the default state of the SUT and many operations of the SUT leave it in a different state from that in which it started.
- There is a strong temptation to reuse the end state of one test condition as the starting state of the next by combining them into a single test because this makes it more efficient.
- This is not recommended because when one assertion fails, the rest of the test is not executed.
  - This makes it hard to achieve Defect Localization.

Test Concerns Separately

- The behavior of a complex application is made up of the aggregate of a large number of smaller behaviors.
- Sometimes, several of these behaviors are provided by the same component.
- Each of these behaviors is a different concern and may have a significant number of scenarios in which it needs to be verified.
- The problem with testing several concerns in a single Test Method is that it will be broken whenever any of the tested concerns is modified.
- Even worse, it won't be obvious which concern is the one at fault.



unittest

unittest

> **`unittest`** has been built into the Python standard library since version 2.1.

> **`unittest`** contains both a testing framework and a test runner.

> **`unittest`** has some important requirements for writing and executing tests.

- You put your tests into classes as methods
- You use a series of special assertion methods in the **`unittest.TestCase`** class instead of the built-in assert statement

unittest

> To convert the earlier example to a **`unittest`** test case, you would have to:

1. Import **`unittest`** from the standard library
2. Create a class called **`TestSum`** that inherits from the **`TestCase`** class
3. Convert the test functions into methods by adding **`self`** as the first argument
4. Change the assertions to use the **`self.assertEqual()`** method on the **`TestCase`** class
5. Change the command-line entry point to call **`unittest.main()`**

Exercise (1/2)

```
import unittest
import coverage

class AccountTest(unittest.TestCase):
    def setUp(self):
        self.account = Account("TR1", 1000)
        self.assertEqual("TR1", self.account.iban)
        self.assertEqual(1000, self.account.balance)
        self.assertEqual(f"Account[ iban: TR1, balance: 1000]", str(self.account))

    def test_withdrawAllAmountThenSuccess(self):
        self.account.withdraw(1000)
        self.assertEqual(0, self.account.balance)

    def test_withdrawNegativeAmountThenFail(self):
        with self.assertRaises(ValueError) as e:
            self.account.withdraw(-1)
            self.assertEqual("amount must be positive.", str(e))
        self.assertEqual(1000, self.account.balance)
```

Exercise (2/2)

```
def test_withdrawOverAmountThenFail(self):
    with self.assertRaises(InsufficientBalance) as e:
            self.account.withdraw(1001)
            self.assertEqual(1, e.deficit)
            self.assertEqual(f"InsufficientBalance[ message: balance is less \
                             than amount., deficit: 1]", str(e))
     self.assertEqual(1000, self.account.balance)

 def test_depositAnyPositiveAmountThenSuccess(self):
     self.account.deposit(1)
     self.assertEqual(1001, self.account.balance)

 def test_depositNegativeAmountThenFail(self):
     with self.assertRaises(ValueError):
         self.account.deposit(-1)
     self.assertEqual(1000, self.account.balance)
```

How to Write Assertions

Method	Equivalent to
.assertEqual(a, b)	a == b
.assertTrue(x)	bool(x) is True
.assertFalse(x)	bool(x) is False
.assertIs(a, b)	a is b
.assertIsNone(x)	x is None
.assertIn(a, b)	a in b
.assertIsInstance(a, b)	isinstance(a, b)

Executing Test Runners

> The Python application that executes your test code, checks the assertions, and gives you test results in your console is called the ***test runner***.

> At the bottom of your test code, add the following code:

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

> This is a command line entry point.

– if you execute the script alone by running python test.py at the command line, it will call **`unittest.main()`**.

– This executes the test runner by discovering all classes in this file that inherit from **`unittest.TestCase`**.

Executing Test Runners

```
python -m unittest test.test1
.
----------------------------------------------------------------
Ran 5 tests in 0.002s

OK
```

Executing Test Runners

```
python -m unittest -v test.test1
test_depositAnyPositiveAmountThenSuccess (test.test1.AccountTest) ... ok
test_depositNegativeAmountThenFail (test.test1.AccountTest) ... ok
test_withdrawAllAmountThenSuccess (test.test1.AccountTest) ... ok
test_withdrawNegativeAmountThenFail (test.test1.AccountTest) ...ok
test_withdrawOverAmountThenFail (test.test1.AccountTest) ... ok

----------------------------------------------------------------------
Ran 5 tests in 0.003s
```

Executing Test Runners

> Instead of providing the name of a module containing tests, you can request an auto-discovery

```
python -m unittest discover -v
test_depositAnyPositiveAmountThenSuccess (test.test1.AccountTest) ... ok
test_depositNegativeAmountThenFail (test.test1.AccountTest) ... ok
test_withdrawAllAmountThenSuccess (test.test1.AccountTest) ... ok
test_withdrawNegativeAmountThenFail (test.test1.AccountTest) ...ok
test_withdrawOverAmountThenFail (test.test1.AccountTest) ... ok

----------------------------------------------------------------------
Ran 5 tests in 0.003s
```

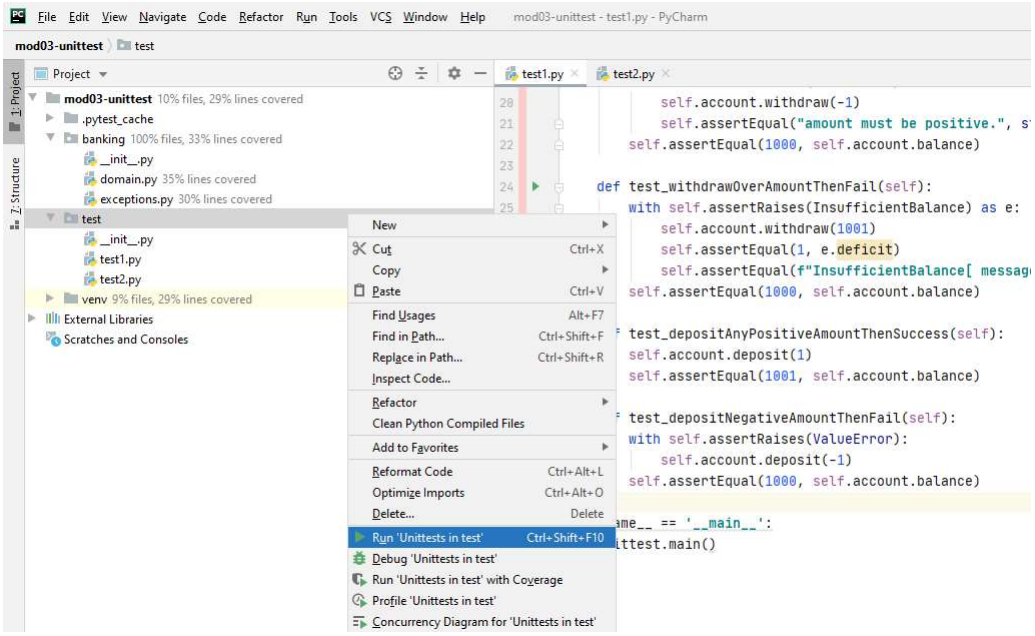
Executing Test Runners

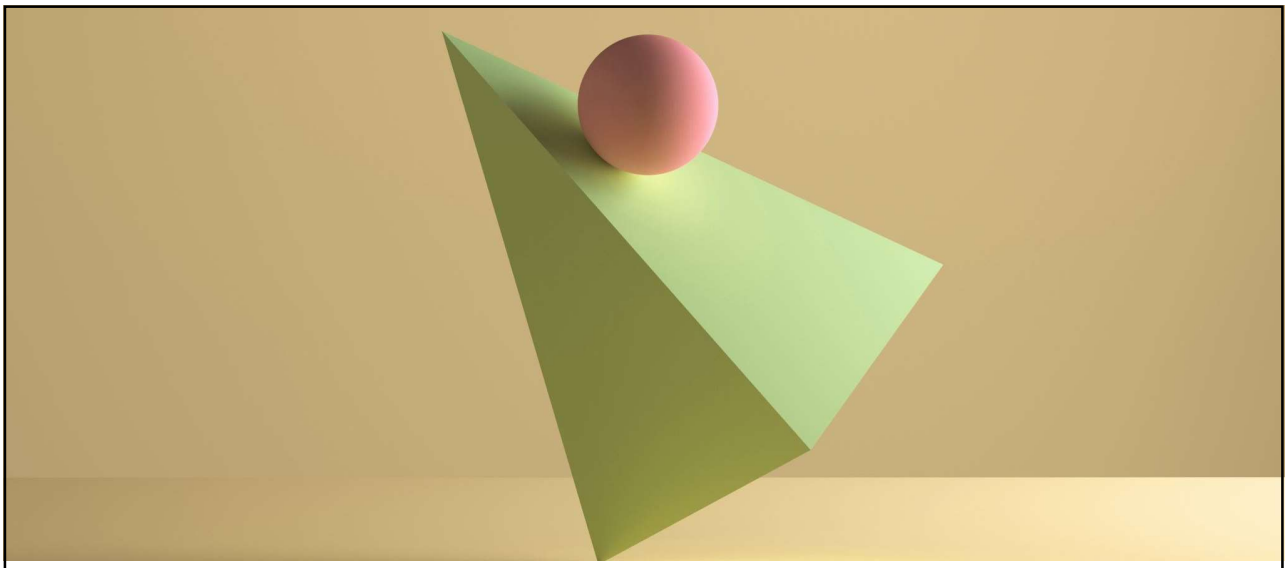
> Once you have multiple test files, as long as you follow the test*.py naming pattern, you can provide the name of the directory instead by using the **-s** flag and ***the name of the directory***:

```
python -m unittest discover -s test -v
test_depositAnyPositiveAmountThenSuccess (test.test1.AccountTest) ... ok
test_depositNegativeAmountThenFail (test.test1.AccountTest) ... ok
test_withdrawAllAmountThenSuccess (test.test1.AccountTest) ... ok
test_withdrawNegativeAmountThenFail (test.test1.AccountTest) ...ok
test_withdrawOverAmountThenFail (test.test1.AccountTest) ... ok

----------------------------------------------------------------------
Ran 5 tests in 0.003s
```

Running Your Tests from **PyCharm**





PyTest

Introduction

- Pytest is a python-based testing framework
- You can write simple to complex tests
  - API Test
  - Database Test
  - UI Test

Advantages of Pytest

> The advantages of Pytest are
  - Can run multiple tests in parallel, which reduces the execution time of the test suite.
  - Has its own way to detect the test file and test functions automatically, if not mentioned explicitly.
  - Allows us to skip a subset of the tests during execution.
  - Allows us to run a subset of the entire test suite.
  - Free and open source.
  - Very easy to start with because of its simple syntax

Pytest - Environment Setup

> To install the version 2.9.1

```
pip install pytest == 2.9.1
```

> To install the latest version

```
pip install pytest
```

Identifying Test files and Test Functions

- Running pytest without mentioning a filename will run all files of format test_*.py or *_test.py in the current directory and subdirectories.
  - Pytest automatically identifies those files as test files.
- We can make pytest run other filenames by explicitly mentioning them.
- Pytest requires the test function names to start with **`test`**.
  - Function names which are not of format test* are not considered as test functions by pytest.
  - We cannot explicitly make pytest consider any function not starting with test as a test function.

Starting With Basic Test

```
import math

def test_sqrt():
    num = 25
    assert math.sqrt(num) == 5

def testsquare():
    num = 7
    assert 7*7 == 40

def tesequality():
    assert 10 == 11
```

Fixture

```
@pytest.fixture
def an_active_account():
    return Account("TR1", 1000, AccountStatus.ACTIVE)


@pytest.fixture
def a_closed_account():
    return Account("TR2", 1000, AccountStatus.CLOSED)
```

Writing a test method

```
def test_withdraw_all_balance_should_success(an_active_account):
    an_active_account.withdraw(1000)
    assert an_active_account.balance == 0
```

pytest.raises

```
@pytest.fixture
def an_active_account():
    return Account("TR1", 1000, AccountStatus.ACTIVE)
```

```
def test_withdraw_should_raise_error(an_active_account):
    with pytest.raises(InsufficientBalanceError):
        an_active_account.withdraw(1001)
```

Parameterized Tests

```
@pytest.fixture
def an_active_account():
    return Account("TR1", 1000, AccountStatus.ACTIVE)
```

```
test_success_amounts = [
    (0.1, 1000.1), (1, 1001), (100, 1100)
]
```

```
@pytest.mark.parametrize("amount,expected", test_success_amounts)
def test_deposit_should_success(an_active_account, amount, expected):
    an_active_account.deposit(amount)
    assert an_active_account.balance == expected
```

Mocking

```
def test_get_customer_account_is_success(a_bank, mocker):
    jack = a_bank.create_customer("1", "jack bauer")
    account = Account("TR1", 1000)
    mocker.patch('banking.Customer.get_account', return_value=account)
    returned_account = a_bank.get_account("TR1")
    assert returned_account == account
    assert returned_account.iban == "TR1"
```