Programming 2

Dries Decuyper
Serhat Erdogan
Janne Gilis
Bart Thumas

Testing

AGENDA

- Introduction
- Pytest
- Parameterized & Approx
- Exceptions & Expected Values
- Setup & Fixtures & Factories
- Assertions

Introduction

CODE CORRECTNESS

- Code correctness can be achieved by
  - Static typing
  - Verification tools
  - Assertions
  - Correctness proofs
  - Writing tests
- Each with their strengths and weaknesses
- Combining them is often possible (Swiss cheese model)

GOOD TESTS

- What makes a good test?
  - Automation
  - Fine grained test
  - Readable test
  - Fast running test
  - Isolated test

AUTOMATION

```
def square(x):
    return x ** 2

print(square(5))
print(square(77))
```

```
def square(x):
    return x ** 2

assert square(5) == 25
assert square(77) == 5929
assert square(25) == 123
# result: AssertionError
```

- Manual testing is a bad approach, often removed
  - Tests are exhaustive, code 100% bug free, no need to test again
  - Code never to be modified

FINE GRAINED TEST

- Tests should return an overview of what went wrong, but also what went right
  - An AssertionError doesn’t give much information
- Tests should only be able to fail for one reason only

READABLE TEST

- Be able to see what exactly was being tested
  - State of the objects
  - The performed actions
  - The expected result and the actual one

FAST RUNNING TEST

- Tests run quite often
  - Tests run after every little code change
  - Can also be run live, continuously in the background while coding
  - While developing, often run before building, committing, deploying, …
    - Maven, Git, CI/CD

ISOLATED TEST

- Tests should run in isolation
  - Not be able to affect each other’s result
  - Order should not matter
  - Run in parallel without affecting the outcome

Pytest

- Pytest configuration
- AssertionError

PYTEST CONFIGURATION

- Pytest runs tests in tests.py because configured so
  - pytest.ini

```
python_files =
    tests.py
    *_tests.py
```

- Pytest collects all functions that start with test, default setting
- A tests that returns normally is considered to have passed. You need to throw an exception in order to make a test fail

ASSERTIONERROR

```
def test_1():
    actual = [1, 2, 3]
    expected = [1, 2, 4]
    assert expected == actual

'''
    def test_1():
        actual = [1, 2, 3]
        expected = [1, 2, 4]
>       assert expected == actual
E       assert [1, 2, 4] == [1, 2, 3]
E         At index 2 diff: 4 != 3
E         Use -v to get more diff

tests.py:4: AssertionError
'''
```

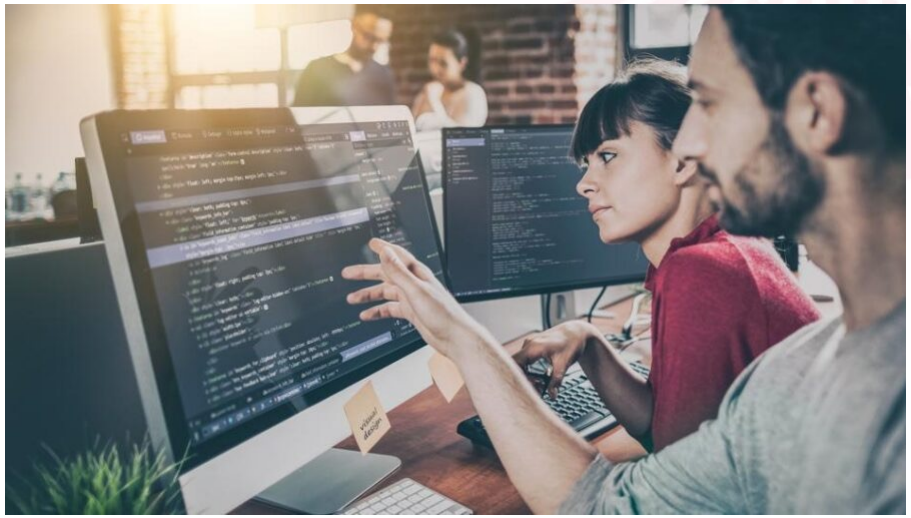
```
def test_2():
    actual = [1, 2, 3]
    expected = [1, 2, 4]
    if actual != expected:
        raise AssertionError()

'''
    def test_2():
        actual = [1, 2, 3]
        expected = [1, 2, 4]
        if actual != expected:
>           raise AssertionError()
E           AssertionError

tests.py:5: AssertionError
'''
```

- While you can throw an assertion manually, we strongly suggest to rely on assert

EXERCISE

- Try the following exercises

- 01-basics



Parameterized & Approx

PARAMETERIZED

```
import pytest

@pytest.mark.parametrize('interval1, interval2', [
    ((1, 5), (3, 6)),
    ((1, 5), (5, 6)),
    ((1, 10), (3, 6)),
    ((6, 8), (3, 6)),
    ((5, 7), (4, 8)),
])
def test_overlapping_intervals(interval1, interval2):
    assert overlapping_intervals(interval1, interval2), f"Interval
{interval1} overlaps with interval {interval2}"
```

- You can equip the assert with an error message
- The parametrize decorator takes two parameters
  - A string with the parameter names, must be the same as the test function's parameters
  - A list of tuples of values to be assigned to the parameters

APPROX

- Try following example, depending your python version this might result in False or True

```
sum([0.1] * 10) == 1
```

- We must take this in consideration while writing tests

```
expected = 1
actual = sum([0.1] * 10)

assert abs(expected - actual) < 0.0001
```

- Or use the approx helper function with optional tolerance

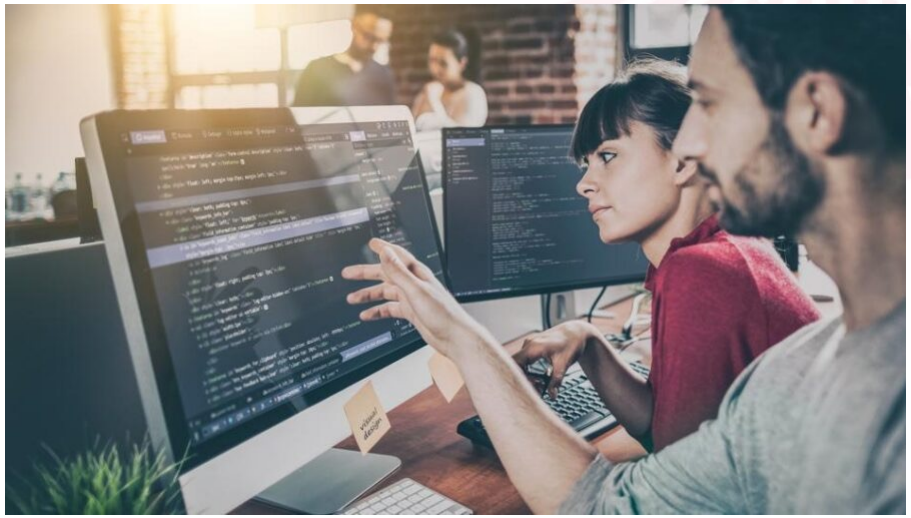
```
from pytest import approx

expected = 1
actual = sum([0.1] * 10)

assert approx(expected, abs=0.1) == actual
```

```
from pytest import approx
from math import pi

assert approx(3.14159265) == pi
assert approx(3.14, abs=0.01) == pi
```

EXERCISE

- Try the following exercises

- 02-parameterized
- 03-approx



Exceptions & Expected Values

- Exceptions
- Expected Values

EXCEPTIONS

- Use the following construct to catch exceptions

```
import pytest

class Animal:
    def __init__(self, name):
        self.name = name

    @property
    def name(self):
        return self.__name

    @name.setter
    def name(self, name):
        if name is None:
            raise ValueError
        self.__name = name

def test_raises_exception():
    with pytest.raises(ValueError):
        animal = Animal(None)
```

- Keep the code inside the with block to a minimum

EXPECTED VALUES

```
from pytest import pytest, approx
from math import sqrt

@pytest.mark.parametrize('n, expected', [
    (1, 1),
    (4, 2),
    (9, 3),
    (16, 4),
    (100, 10),
    (2, 1.41421356),
    (5785, 76.059187)
])
def test_sqrt(n, expected):
    actual = sqrt(n)
    assert approx(expected) == actual
```

```
from pytest import pytest, approx
from math import sqrt

@pytest.mark.parametrize('n, expected', [
    (x * x, x) for x in [1, 2, 3, 4, 10,
1.41421356, 76.059187]
])
def test_sqrt(n, expected):
    actual = sqrt(n)
    assert approx(expected) == actual
```

- We could start off with the expected value, square it and then specify that sqrt() of the squared value should return the expected value

EXPECTED VALUES

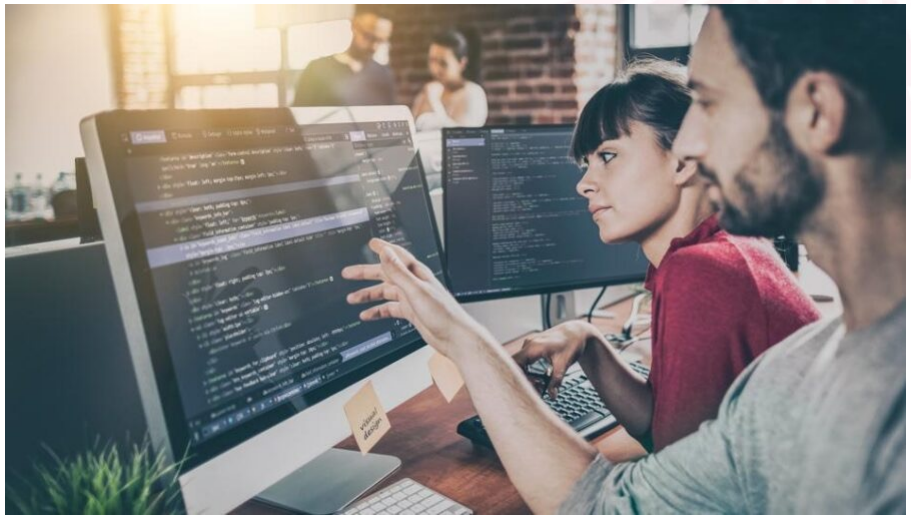
```
from pytest import pytest, approx
from math import sqrt

@pytest.mark.parametrize('n, expected', [
    (x * x, x) for x in (x * 0.73 for x in
range(1000))
])
def test_sqrt(n, expected):
    actual = sqrt(n)
    assert approx(expected) == actual
```

- We could easily increase the number of inputs and add some extra computation

EXERCISE

- Try the following exercises

- 04-exceptions
- 05-expected-values



Setup & Fixtures & Factories

- Setup
- Fixutres
- Factories

SETUP

```
def setup_function():
    ...

def teardown_function():
    ...

def test_1():
    ...

def test_2():
    ...

def test_3():
    ...
```

Called in following order:

- setup_function()
- test_1()
- teardown_function()
- setup_function()
- test_2()
- teardown_function()
- setup_function()
- test_3()
- teardown_function()

- Pytest can also looks for a function named setup_function. Before running a test, Pytest will call setup_function
- Similarly, you can also have a function named teardown_function. This will be called after each test

FIXTURES

- Having a large setup_function has two major disadvantages
  - The more a setup_function does, the more chance it has to fail
  - It slows things down unnecessarily
- A better solution is to rely on Pytest's @fixture functionality
  - Tests (and fixtures) can declare their dependencies using parameters
  - Pytest will then automatically call the corresponding @fixture and pass its return value as arguments
  - A test/fixture can have more dependencies
  - @fixtures solve the shortcomings mentioned above

FIXTURES EXAMPLE

```
from datetime import date, timedelta
import pytest

class CalendarStub:
    def __init__(self, today):
        self.today = today

@pytest.fixture
def today():
    return date(2000, 1, 1)

@pytest.fixture
def tomorrow(today):
    return today + timedelta(days=1)

@pytest.fixture
def calendar(today):
    return CalendarStub(today)


def test_calendar_stuff(calendar):
    assert(calendar.today == date(2000, 1, 1))
```

FACTORIES DESCRIPTIVE NAMES

```python
import pytest
from datetime import date, timedelta

# More code

@pytest.fixture
def task(tomorrow):
    return Task('xxx', tomorrow)

def test_task_creation(task, tomorrow):
    assert task.description == 'xxx'
    assert task.due_date == tomorrow
    assert task.finished == False
```

```python
import pytest
from datetime import date, timedelta

# More code

@pytest.fixture
def unfinished_task_due_by_tomorrow(tomorrow):
    return Task('xxx', tomorrow)

def test_task_creation(unfinished_task_due_by_tomorrow, tomorrow):
    assert unfinished_task_due_by_tomorrow.due_date == tomorrow
    assert unfinished_task_due_by_tomorrow.finished == False
```

- Lack of readability and the test isn't self contained
- Use a more descriptive variable name
  - Why not use unfinished_task_due_by_tomorrow_described_as_xxx
  - Approach becomes very impractical, very quickly

FACTORIES MUTABLE FIXTURES

```
import pytest
from datetime import date, timedelta

# More code

def test_task_becomes_finished(sut, unfinished_task):
    sut.add_task(unfinished_task)

    unfinished_task.finished = True

    assert [] == sut.due_tasks
    assert [unfinished_task] == sut.finished_tasks
```

- Misleading names ought to be avoided at all costs
  - Variable's name should convey all important information
  - A basic variable naming rule says to pick a name that always describes the value accurately

FACTORY FUNCTIONS

```
def create_unfinished_task():
    return Task('xxx', date(2000, 1, 1))

def test_task_becomes_finished(sut):
    task = create_unfinished_task()
    sut.add_task(task)

    task.finished = True

    assert [] == sut.due_tasks
    assert [task] == sut.finished_tasks
```

- Same advantages as @fixture
- The factory function has an accurate and precise name:
  - Makes clear that the returned task is unfinished
  - Doesn't convey any extra information
  - Variable name at no point contradicts the variable's value.

FACTORY FUNCTIONS FLEXIBLE

```
def create_task(*, description='default description',
due_date=None, finished=False):
    due_date = due_date or date(2000, 1, 1)
    task = Task(description, due_date)
    if finished:
        task.finished = True
    return task
```

```
def test_task_becomes_finished(sut):
    task = create_task()
    sut.add_task(task)

    task.finished = True

    assert [] == sut.due_tasks
    assert [task] == sut.finished_tasks

def test_task_is_finished(sut):
    task = create_task(finished=True)
    sut.add_task(task)

    assert [] == sut.due_tasks
    assert [task] == sut.finished_tasks
```

- A single parametrized factory function might be preferable over distinct factory functions (create_finished_task, create_unfinished_task, create_task_due_tomorrow, etc.)
- The * in create_task's parameter list forces callers to use keyword arguments

FACTORY FUNCTIONS MORE CODE

```
import pytest
from datetime import date, timedelta

class Task:
    def __init__(self, description, due_date, finished=False):
        self.description = description
        self.due_date = due_date
        self.finished = finished

    @property
    def finished(self):
        return self._finished

    @finished.setter
    def finished(self, value):
        self._finished = value
        if value:
            system = SystemUnderTest.get_instance()
            if self in system.due_tasks:
                system.due_tasks.remove(self)
            system.finished_tasks.append(self)
```

```
class SystemUnderTest:
    _instance = None

    def __init__(self):
        if not SystemUnderTest._instance:
            SystemUnderTest._instance = self
        self.due_tasks = []
        self.finished_tasks = []

    @classmethod
    def get_instance(cls):
        if not cls._instance:
            cls._instance = SystemUnderTest()
        return cls._instance

    def add_task(self, task):
        if task.finished:
            self.finished_tasks.append(task)
        else:
            self.due_tasks.append(task)
```

- More code for previous slides

FACTORY FUNCTIONS MORE CODE

```
@pytest.fixture
def today():
    return date(2000, 1, 1)

@pytest.fixture
def tomorrow(today):
    return today + timedelta(days=1)

@pytest.fixture
def sut():
    return SystemUnderTest()

def create_task(*, description='default description',
due_date=None, finished=False):
    due_date = due_date or date(2000, 1, 1)
    task = Task(description, due_date)
    if finished:
        task.finished = True
    return task
```

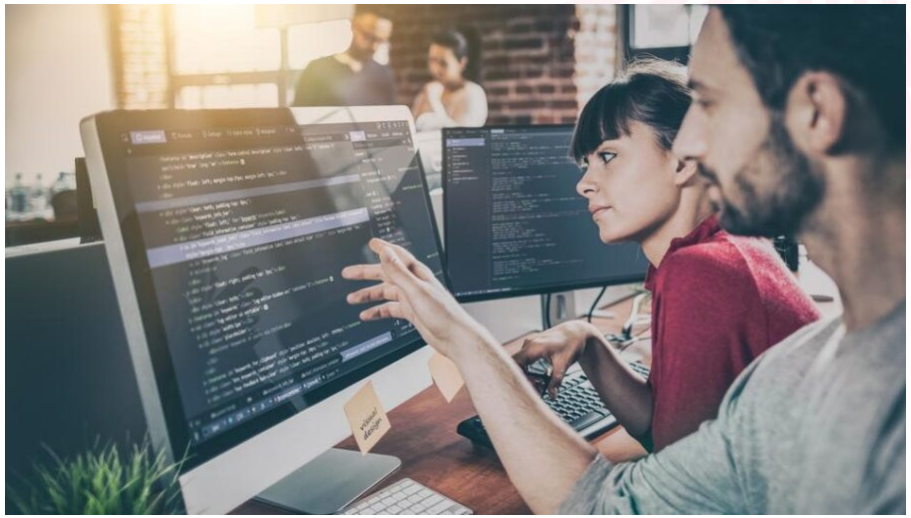
```
def test_task_becomes_finished(sut):
    task = create_task()
    sut.add_task(task)

    task.finished = True

    assert [] == sut.due_tasks
    assert [task] == sut.finished_tasks

def test_task_is_finished(sut):
    task = create_task(finished=True)
    sut.add_task(task)

    assert [] == sut.due_tasks
    assert [task] == sut.finished_tasks
```

- More code for previous slides

EXERCISE

- Try the following exercises

- 10-setup
- 11-fixtures
- 12-factories



Assertions

- Debug vs release build
- Assert

DEBUG VS RELEASE BUILD

- In the debug build, no optimizations are made. The code is compiled in a very straightforward manner, and possibly extra metadata is added to assist you in debugging
- In a release build, all optimizations are turned on and none of the debugging aids are present. This is the build the end user will receive

DEBUG VS RELEASE BUILD

- In the debug build, no optimizations are made. The code is compiled in a very straightforward manner, and possibly extra metadata is added to assist you in debugging
- In a release build, all optimizations are turned on and none of the debugging aids are present. This is the build the end user will receive

DEBUG VS RELEASE BUILD

```
assert False, 'Failure!'
```

```
$ py .\tests.py
Traceback (most recent call last):
  File "C:\Userspy\Downloads\tests.py", line 1, in
<module>
    assert False, 'Failure!'
AssertionError: Failure!
```

```
assert False, 'Failure!'
```

```
$ py -O .\tests.py
```

- The most significant difference between debug and release build in Python is how asserts are executed
  - In release mode they are ignored

ASSERT

```
def max(ns):
    result = 8

    # Check that result is an element
of the given list
    assert result in ns
    # Check that all values in the
given list are not greater than result
    assert all(n <= result for n in ns)

    return result

print(max([1,5,7,3,8,2,9,5,1]))
```

```
def max(ns):
    result = 4

    # Check that result is an element of
the given list
    assert result in ns
    # Check that all values in the given
list are not greater than result
    assert all(n <= result for n in ns)

    return result

print(max([1,5,7,3,8,2,9,5,1]))
```

```
def max(ns):
    result = ns[0]
    for n in ns:
        if n > result:
            result = n

    # Check that result is an element
of the given list
    assert result in ns
    # Check that all values in the
given list are not greater than result
    assert all(n <= result for n in ns)

    return result

print(max([1,5,7,3,8,2,9,5,1]))
```

- Assert can be used in regular code to perform self-checks

EXERCISE

- Try the following exercises

- 13-assertions