Testix

Release 9.0.0

Yoav Kleinberger

Nov 01, 2022

CONTENTS:

CHAPTER

ONE

THE TEST-FIRST MOCKING FRAMEWORK

Testix is the Test-First (TDD) Friendly Mocking framework for Python, meant to be used with pytest

> **Warning:** These docs are under development!

Testix is special because it allows you to specify what your mock objects do, and it then enforces your specifications automatically. It also reduces (albeit not entirely) mock setup.

Other frameworks usually have a flow like this: * setup mock * let code do something with mock * assert mock used in correct way

Testix flow is a bit different * setup mock objects * specify *exactly* what should happen to them using a Scenario context

CHAPTER

TWO

QUICK EXAMPLE

Here is a quick example of how Testix works.

```
# to test the Chatbot class, we pass it a mock socket called "sock"
tested = chatbot.Chatbot(Fake('sock'))

# create a Scenario context
# inside, you specify exactly what the unit should do with the objects its handed
with Scenario() as s:

    s.sock.recv(4096) >> 'request text'  # unit must call sock.recv(4096).
                                         # this call will return 'request text'
    s.sock.send('response text')

    # call your unit's code
    tested.go()


# Scenario context ends, and verifies everything happened exactly as specified
# No more, no less
```

Note that you do not have to setup `sock.recv` or `sock.send` - once `sock` is set up, it will generate other mock objects automatically as you go along with it. Only "top level" mock objects need to be setup explicitly.

The `Scenario` object does essentially two things: * setup our expectations (these are the `s.sock.*` lines) * enforce our expectations (this is done by the `with` statement)

Want to know more? Read the :doc:tutorial.

CHAPTER

THREE

ADVANTAGES

Testix has been written to promote the following

- readability - the expectations are very similar to the actual code that they test (compare `s.sock.recv(4096)` with the standard `sock.recv.assert_called_once_with(4096)`
- Test Driven Development friendliness: if you use `sock.recv.assert_called_once_with(4096)`, you must use it after the code has run. With Testix, you specify what you *expect*, and the asserting is done for you by magic.

What are you waiting for? Read the *Tutorial*

CHAPTER

FOUR

READ MORE

4.1 Tutorial

This tutorial will walk you through Test Driven Development using Testix and pytest.

We will develop a small project in this tutorial, test driven of course, which solves the following real-life problem: suppose you want to run some subprocess, and you want to read its output line-by-line in real time and take appropriate action.

An example application might be that you want to monitor live logs and do something whenever a log line has `ERROR` in it.

Let's call this library `LineMonitor`.

4.1.1 Design of the LineMonitor

Python has an excellent library called subprocess, which allows a quite generic inteface for launching subprocsses using its Popen class.

We want to have a `LineMonitor` class which:

- will launch subprocesses using `subprocess` under the hood
- will allow the caller to register callbacks that get called from every line of output from the subprocess
- will also implement an iterator form, e.g. you can write something like

```
for line in line_monitor:
    print(f'this just in: {line}')
```

Since this is a Test Driven Development tutorial as well as a Testix tutorial, let's discuss the tests.

First a short primer on types of tests.

Unit Tests

Unit tests check that each unit of code (usually a single class or module) performs the correct business logic.

Generally speaking, unit tests

- test logic
- do not perform I/O (perhaps only to local files)
- use mocks (not always, but many times) - this is where Testix comes in.

Integration Tests

Integration tests test that various "units" fit together.

Generally speaking, integration tests

- perform some actual I/O
- do not rigorously test logic (that's the unit test's job)

End-to-End (E2E) Tests

In our case, since the project is quite small, the integration test will actually test the scope of the entire project. and so it is more appropriately called an End-to-End (E2E) Test.

In real projects, E2E tests usually include * an actual deployment, which is as similar as possible to real life deployments. * various UI testing techniques, e.g. launching a web-browser to use some webapp

In our toy example, we don't have such complications.

Let's move on.

4.1.2 End-to-End Test

When we say "Test First" - this means that we go about thinking about our code by thinking about how to test that it works.

When we say "Test Driven" - this means that we let our thinking about tests *define* how the code will work.

In this way, the tests *drive* our development.

So, how should we go about testing that everything works in our `LineMonitor` library? obviously, we should launch a subprocess which known output, and see that we can get all the lines emitted into a callback which we define.

So we want our users to do something like this

```
import line_monitor.monitor

captured_lines = []

monitor = line_monitor.LineMonitor(['ls', '-l'], on_output=captured_lines.append) #␣
↪launch `ls -l` to list the files, lines get appended into our captured_lines list
monitor.monitor() # monitor process until it ends
for line in captured_lines:
    print(f'saw this: {line}')
```

Now that we have a rough idea, let's write a test which will make this precise. The code below is not the final test, and will not really work, but it's a sketch:

Listing 1: end-to-end (E2E) test

```
import line_monitor

def test_line_monitor():
    captured_lines = []
    tested = line_monitor.LineMonitor()
    tested.register_callback(captured_lines.append)
    PRINT_10_LINES_COMMAND = ['python', '-c', 'for i in range(10): print(f"line {i}")']
    tested.launch_subprocess(PRINT_10_LINES_COMMAND)
    tested.monitor()
    EXPECTED_LINES = [f'line {i}' for i in range(10)]
    assert captured_lines == EXPECTED_LINES
```

What do we have here? We create the tested object, a `LineMonitor` object called `tested`. We provide it a callback (which is just the `.append` method on the `captured_lines` list). We then tell it to launch the subprocess with similar arguments to `subprocess.Popen` - and we give it a specific subprocess which we know will print 10 lines of output. Finally, after the subprocess has ended we test that the captured output is what we expect it to be.

Tests Driving our Code

Note that in the process of developing the *test*, we chose the names of various API calls, e.g. `launch_subprocess` (we could have launched the subprocess in the constructor like in the draft we wrote before, but it felt more natural to me to separate the creation of a `LineMonitor` object from actual launching of a subprocess).

This is what we mean when we say that tests *drive* development.

However, to truly work Test Driven - we need to make this test *fail properly*.

4.1.3 Failing Properly

For convenience, here again is our *End-to-End Test*:

Listing 2: end-to-end (E2E) test

```
import line_monitor

def test_line_monitor():
    captured_lines = []
    tested = line_monitor.LineMonitor()
    tested.register_callback(captured_lines.append)
    PRINT_10_LINES_COMMAND = ['python', '-c', 'for i in range(10): print(f"line {i}")']
    tested.launch_subprocess(PRINT_10_LINES_COMMAND)
    tested.monitor()
    EXPECTED_LINES = [f'line {i}' for i in range(10)]
    assert captured_lines == EXPECTED_LINES
```

If we run it will of course not work:

```
$ python -m pytest docs/line_monitor/tests/e2e
```

Results ultimately in

```
E   ModuleNotFoundError: No module named 'line_monitor'
```

That is because none of the code for `line_monitor` exits yet. This is a sort of failure, but it's not very interesting.

What we want is for the test to fail *properly* - we want it to fail *not* because our system doesn't exist - we want it to fail because our system does not implement the correct behaviour yet.

In concrete terms, we want it to fail because a subprocess has seemingly been launched, but its output has not been captured by our monitor. In short, we want it to fail on our `assert` statements, not due to some technicalities

So, let's write some basic code that achieves just that. We create a `line_monitor.py` file within our `import` path with skeleton code:

Listing 3: line_monitor.py

```
class LineMonitor:
    def register_callback(self, callback):
        pass

    def launch_subprocess(self, *popen_args, **popen_kwargs):
        pass

    def monitor(self):
        pass
```

Now if we run the test:

```
$ python -m pytest docs/line_monitor/tests/e2e
```

We get a proper failure

```
....... OUTPUT SKIPPED FOR BREVITY .......

>       assert captured_lines == EXPECTED_LINES
E       AssertionError: assert [] == ['line 0', 'l...'line 5', ...]
```

This is a **proper failure** - the test has done everything right, but the current `LineMonitor` implementation does not deliver on its promises.

The Importance of Proper Failure

Congratulations, we have a **failing test**! This is the first milestone when developing a feature using Test Driven Development. Let's briefly explain whey this is so important, and why this is superior to writing tests for previously written, already working code.

Essentially, imagine we wrote a test, wrote some skeleton code, ran the test - and it *didn't fail*. Well, that would obviously mean that our test was bad. This is admittedly rare, but I've seen it happen.

The more common scenario however is that we wrote the test, wrote the skeleton code, ran the test - and it failed, but *not in the way we planned*. This means that the test does not, in fact, test what we want.

If we write our test *after* we've developed our code, how will we ever know that the test actually tests what we think it is testing? You'd be amazed at the number of tests which exist out there and in fact, do not test what they are supposed to.

I have seen with my own eyes, many times, tests that do not test *anything at all*. This happens because once code has been written, the tests are written to accommodate the code, which is *exactly* the opposite of what should happen.

The last point is super important so I will rephrase it in a more compelling way: think about testing the performance of a human being, not a computer program, e.g. testing a student in high school or university. Should we have the student write his or her answers first, and *then* write the test to accommodate these answers? *Utterly absurd*.

We should write the test first, and then use it to test the student.

If we write our tests first, and **fail them properly**,

- we make sure they actually test what they pretend to to
- we think hard about how to test this functionality - we gain focus on what our software is supposed to do
- we take pains to actually think about how the code will be use: we let the test drive the design of the application

So, it is essential before developing some behaviour, that our tests fail properly.

Now, let's get on with implementing the `LineMonitor`. This will require - surprise - some more tests - unit tests, which is what Testix is all about.

4.1.4 Testix Basics

It's time to start writing unit tests using Testix. In this section we'll cover all the basics, then move on to our more complete `LineMonitor` example.

Working With Scenarios and Fake Objects

In this part we introduce the basic building blocks of writing a unit test with Testix. As mentioned in *Unit Tests*, this is where we test our business logic: the required behaviour and edge cases. To control carefully how our code interacts with the outside world, we use so called Mock objects or as they are sometimes called Fake objects.

Before seeing how Testix does it, let's review the concept of Mock objects.

Mock Objects

Mock Objects are objects that simulate some object that our code needs to interact with, that we want to test carefully. As an example - suppose our code needs to send data over a socket, which it receives as a parameter called `sock`

```
send_some_data(sock, b'the data')
```

When testing we

- don't *really* want to send data over a *real* socket
- do want to verify that the `send_some_data` function called `sock.send(b'the data')`.

The solution is to pass `send_some_data` an object that implements a `.send` method, but which is not an actual socket. Instead this object will just record that `.send` was called, and we'll be able to query it to see that it was called with `b'the data'`. The idea here is that there's no point testing sockets - we know that those work. The point here is to test that *our code does the right thing with the socket*.

The Standard Library Way - unittest.mock

The approach taken by the standard unittest.mock module from the Python standard library, is to provide us with a generic `Mock` class which records every call that is made to it, and has some helper functions to help as assret that some things happened.

```
import unittest.mock

def test_sending_data():
    sock = unittest.mock.Mock()
    send_some_data(sock, b'the data')
    sock.send.assert_called_with(b'the data') # this verifies that `send_some_data` did␣
↪the right thing
```

Let's see how Testix approaches the same idea. We will discuss the advantages of the Testix way later on.

Testix Fake Objects and Scenarios

Setting the Expectations

We'll start by introducing a test for `send_some_data` and then explaining it.

Note that first we need to fail the test - so `send_some_data` here is only a skeleton implementation that really does nothing.

Listing 4: test_sending_data.py

```
1  from testix import *
2
3  import data_sender
4
5  def test_sending_data():
6      fake_socket = Fake('sock')
7      with Scenario() as s:
8          s.sock.send(b'the data')
9
10         data_sender.send_some_data(fake_socket, b'the data')
```

Listing 5: data_sender.py skeleton implementation

```
def send_some_data(socket, data):
    pass
```

What's going on here? First, we create a Fake object `sock` - this is Testix generic mock object - note that we define a name for it explicitly - `'sock'`. We then start a `Scenario()` context manager in the `with Scenario() as s` statement.

A Scenario is a way to specify the required behaviour - what do we demand happen to our fake objects? In this case, we specify one demand:

```
s.sock.send(b'the data')
```

This means - we expect that the Fake object method `sock.send` be called with `b'the data'` as the argument. When the `Scenario` context ends - the `Scenario` object will automatically enforce these expectations, as we'll see shortly.

Finally - we cannot hope to meet the demands of the test without actually calling the code:

```
send_some_data(fake_socket, b'the data')
```

Let's try to run this test. Of course we expect failure - the `send_some_data` function does not, after all, send the data.

```
$ python -m pytest -v docs/tutorial/other_tests/data_sender_example/1


....... OUTPUT SKIPPED FOR BREVITY .......
E       Failed:
E       testix: ScenarioException
E       testix details:
E       Scenario ended, but not all expectations were met. Pending expectations␣
↪(ordered): [sock.send(b'the data')]
```

As you can see, Testix tells us that "not all expectations were met", and details the missing expectation in a list: `sock.send(b'the data')`.

We have a **properly failing test**, yay!

Meeting the Expectations

Now that we know that the test's expecations aren't being met - let's change the code to meet them:

Listing 6: meet the demand for sending data

```
def send_some_data(socket, data):
    socket.send(data)
```

Now our tests pass

```
python -m pytest -v docs/tutorial/other_tests/data_sender_example/2

docs/tutorial/other_tests/data_sender_example/2/test_sending_data.py::test_sending_data␣
↪PASSED
```

Yay :)

Let's say that now we want our sending function to send a specific header before the data which specifies the data's length. Since we're doing TDD here, we first set our expectations in the test

Listing 7: testing for a header

```
1 from testix import *
2
3 import data_sender
4
5 def test_sending_data():
6     fake_socket = Fake('sock')
7     with Scenario() as s:
8         s.sock.send(b'SIZE:8 ')
```

(continues on next page)

(continued from previous page)

```
        s.sock.send(b'the data')

        data_sender.send_some_data(fake_socket, b'the data')
```

Now our scenario demands that send() be called twice - once with the header, and then with the data.

Next move - let's see that our test fails properly. When we run it we get

```
E       Failed:
E       testix: ExpectationException
E       testix details:
E       === Scenario (no title) ===
E        expected: sock.send(b'SIZE:8 ')
E        actual  : sock.send(b'the data')
E       === OFFENDING LINE ===
E        socket.send(data) (/home/yoav/work/testix/docs/tutorial/tests/data_sender.py:2)
E       === FURTHER EXPECTATIONS (showing at most 10 out of 1) ===
E        sock.send(b'the data')
E       === END ===
```

What happened here? Well, the scenario wants to see `sock.send(b'SIZE:8 ')` - however, since we have not changed our code yet, the actual call is the good old `sock.send(b'the data')`, therefore the *expected* call is different from the *actual* call, and Testix fails the test for us. It also specifies the particuar line that got us in trouble, and gives us a peek into the next expecations in the scenario.

Good news, we have a properly failing test. Now, let's meet the demands:

Listing 8: sending a header 1

```

def send_some_data(socket, data):
    length = len(data)
    header = b'SIZE:' + bytes(str(length), encoding='latin-1')
    socket.send(header)
    socket.send(data)
```

OK let's go:

```
E       Failed:
E       testix: ExpectationException
E       testix details:
E       === Scenario (no title) ===
E        expected: sock.send(b'SIZE:8 ')
E        actual  : sock.send(b'SIZE:8')
E       === OFFENDING LINE ===
E        socket.send(header) (/home/yoav/work/testix/docs/tutorial/tests/data_sender_
↪prefix.py:5)
E       === FURTHER EXPECTATIONS (showing at most 10 out of 1) ===
E        sock.send(b'the data')
```

Oops! Seems like we forgot a `b' '`, let's correct our code:

Listing 9: sending a header 2

```

def send_some_data(socket, data):
    length = len(data)
    header = b'SIZE:' + bytes(str(length), encoding='latin-1') + b' '
    socket.send(header)
    socket.send(data)
```

Now the test passes.

```
$ python -m pytest -v docs/tutorial/other_tests/data_sender_example/prefix_2

docs/tutorial/other_tests/data_sender_example/prefix_2/test_sending_data.py::test_
↪sending_data PASSED
```

More Advanced Tests

Specifying Return Values

Previously we have specified how a `Fake` object named `"sock"` should be used by our code.

When we say `s.sock.send(b'the data')` we express the expectation that the code under test will call the `.send()` method with exactly one argument, whose value should equal exactly `b'the data'`.

When the code does this with `"sock"`'s `.send()` method, however, what value is returned by method call?

The answer in this case is `None` - but Testix also allows us to define this return value. This is useful when you want to test thing related to what function calls on `Fake` objects return, e.g. thing about testing some code that receives data on one socket, and sends the length of said data to another socket.

We therefore expect that there will be a `.recv()` call on one socket which returns some data, this data in turn is converted to a number (its length), which is then encoded and sent on the outgoing socket.

Here's how to test this in Testix

```
from testix import *
import forwarder

def test_forward_data_lengths():
    tested = forwarder.Forwarder()
    incoming = Fake('incoming_socket')
    outgoing = Fake('outgoing_socket')
    with Scenario() as s:
        # we'll require that the length of the data is sent, along with a ' ' separator
        s.incoming_socket.recv(4096)  >>  b'some data'
        s.outgoing_socket.send(b'9 ')
        s.incoming_socket.recv(4096)  >>  b'other data'
        s.outgoing_socket.send(b'10 ')
        s.incoming_socket.recv(4096)  >>  b'even more data'
        s.outgoing_socket.send(b'14 ')

        tested.forward_once(incoming, outgoing)
```

(continues on next page)

(continued from previous page)

```
        tested.forward_once(incoming, outgoing)
        tested.forward_once(incoming, outgoing)
```

We see here a pattern which is common with Testix - specifying an entire scenario of what should happen, then making it happen by calling the code under test.

Here's another version of the same test

```
from testix import *
import forwarder

def test_forward_data_lengths():
    tested = forwarder.Forwarder()
    incoming = Fake('incoming_socket')
    outgoing = Fake('outgoing_socket')
    with Scenario() as s:
        # we'll require that the length of the data is sent, along with a ' ' separator
        s.incoming_socket.recv(4096)  >>  b'some data'
        s.outgoing_socket.send(b'9 ')
        tested.forward_once(incoming, outgoing)

        s.incoming_socket.recv(4096)  >>  b'other data'
        s.outgoing_socket.send(b'10 ')
        tested.forward_once(incoming, outgoing)

        s.incoming_socket.recv(4096)  >>  b'even more data'
        s.outgoing_socket.send(b'14 ')
        tested.forward_once(incoming, outgoing)
```

You should use the style that makes the test most readable to you.

For later reference, here's the code that passes this test:

```
class Forwarder:
    def forward_once(self, read_from, write_to):
        data = read_from.recv(4096)
        binary = bytes(f'{len(data)} ', 'latin-1')
        write_to.send(binary)
```

Exactness

What happens if we now change the code above to read

```
class Forwarder:
    def forward_once(self, read_from, write_to):
        data = read_from.recv(4096)
        binary = bytes(f'{len(data)} ', 'latin-1')
        write_to.send(binary)
        write_to.close()
```

That is, we decided we want to close the outgoing socket for some reason.

```
$ python -m pytest -v docs/tutorial/other_tests/more_advanced/2/test_forward_lengths.py

 ...


    def _fail_py_test( exceptionFactory, message ):
>       return pytest.fail( message )
E       Failed:
E       testix: ExpectationException
E       testix details:
E       === Scenario (no title) ===
E        expected: incoming_socket.recv(4096)
E        actual  : outgoing_socket.close()
E       === OFFENDING LINE ===
E        write_to.close() (/home/yoav/work/testix/docs/tutorial/other_tests/more_
↪advanced/2/forwarder.py:6)
E       === FURTHER EXPECTATIONS (showing at most 10 out of 3) ===
E        outgoing_socket.send(b'10 ')
E        incoming_socket.recv(4096)
E        outgoing_socket.send(b'14 ')
E       === END ===
```

As you can see, the test fails, and the new `.close()` is now, in Testix jargon, the "offending line".

This is because Testix expectations are asserted in an *exact* manner - we define *exactly* what we want, *no more - no less*.

This makes Testix very conducive to Test Driven Development - if you change the code before changing the test - it will probably result in failures. When approaching adding new features - start with defining a test for them.

We'll discuss exactness some more next.

Exact Enforcement

Testix enforces function calls which are specified in a Scenario. It enforces

- the order in which calls are made
- the exact arguments, positional and keyword, which are used
- unexpected calls are considered a failure

Wrong Arguments

So, e.g. if we have a test like this:

```
1 from testix import *
2 import my_code
3
4 def test_my_code():
5     with Scenario() as s:
6         s.source.get_names('all', order='lexicographic', ascending=True) >> ['some',
↪'names']
7         s.destination.put_names(['some', 'names'])
```

(continues on next page)

(continued from previous page)

```
        my_code.go(Fake('source'), Fake('destination'))
```

The code *must* be some variation of

```
names = name_source.get_names('all', order='lexicographic', ascending=True)
# and at some point later...
name_destination.put_names(names)
```

any of these will cause a failure:

```
name_source.get_names('all', 'lexicographic', True)       # lexicographic should be a␣
↪keyword argument
name_source.get_names('all', True, order='lexicographic') # ascending should be a␣
↪keyworkd argument
name_source.get_names(spec='all', order='lexicographic', ascending=True) # spec is␣
↪unexpected
```

Unexpected Calls

This code will also make the test fail:

```
def go(source, destination):
    names = source.get_names('all', order='lexicographic', ascending=True)
    destination.put_names(names)
    destination.something_else()
```

and Testix will report

```
E       Failed:
E       testix: ExpectationException
E       testix details:
E       === Scenario (no title) ===
E       unexpected call: destination.something_else()
E       Expected nothing
```

That is, the `Scenario`'s various expectations were met, but then the code "surprised" Testix with another call on a `Fake` object.

As we said before, the right way to specify Testix Scenarios as to specify what you want *exactly - no more, no less*.

Ways Around Exactness

Sometimes, this exactness is too much - Testix supports ways around this, but most of the time, it is good to be exact. These features are out of the scope of this tutorial, and are documented separately.

Recap

We can now summarize the essentials of the Testix approach:

- use `Fake` objects to simulate various entities
- use a `Scenario` object to define an exact set of demands or *expectations*
- the `Scenario` object not only defines our expectations, it is also used in a `with` statement to *enforce* them.
- return values from `Fake` objects may be specified using >>.

By requiring the developer to define his or her demands using a the Scenario concept, Testix lends itself in particular to Test Driven Development - think about testing first, write the code only after you have exactly defined what you want it to do.

We can also now recognize some major advantages over the mock objects from the Python Standard Library.

- Testix lends itself naturally to the Test Driven approach to development (TDD) through its `Scenario` concept and the "no less - no more" approach that makes it harder to change the functionality of code without changing the test first.
- test syntax is more visually similar to the code under test, e.g. the `s.sock.send(b'the data')` is visually similar to the same as the actual code `socket.send(data)`. This makes tests more readable and easy to understand.
- Whatever expectations you define for you mock objects - they will be *exactly enforced* - defining expectations and enforcing them *is one and the same*.

4.1.5 Line Monitor Unit Tests

We now turn to developing, TDD style, our *LineMonitor* library.

Developing Test Driven style means we add behaviours one by one, for each behaviour we go through the RED-GREEN-REFACTOR loop:

- RED: write a *properly failing test*
- GREEN: write code that passes the test - the code doesn't have to be pretty
- REFACTOR: tidy up the code to make it readable

Sometimes the REFACTOR step is not needed, but we should always at least consider it.

Let's go.

Launching the Subprocess

High Level Design

We will implement `LineMonitor` as follows:

- a `LineMonitor` sill launch the subprocess using the subprocess Python standard library.
- It will attach a pseudo-terminal to said subprocess (using pty). If you don't know too much about what a pseudo-terminal is - don't worry about it, I don't either.

  Essentially it's attaching the subprocess's input and output streams to the father process. Another way of doing this is using pipes, but there are some technical advantages to using a pseudo-terminal.

- it will monitor the terminal using `poll()` from the standard Python library's select module. This call allows to you check if the pseudo-terminal has any data available to read (that is, check if the subprocess has written some output).
- when data is available, we will read it line by line, and send it to the registered callbacks.

Let's start by working on the launching a subprocess with an attached pseudo-terminal.

Implementation

First step is to launch the subprocess with an attached pseudo-terminal. Let's write a test for that. We want to enforce, using Testix, that `subprocess.Popen()` is called with appropriate arguments.

If the following paragraph is confusing, don't worry - things will become clearer after you see it all working.

Since Testix's `Scenario` object only tracks Testix Fake objects, we must somehow fool the `LineMonitor` to use a `Fake('subprocess')` object instead of the actual `subprocess` module. We need to do the same for the `pty` module.

There's more than one way of doing this, but here we will use Testix's helper fixture, `patch_module`.

```
1  from testix import *
2  import pytest
3  import line_monitor
4
5  @pytest.fixture
6  def override_imports(patch_module):
7      patch_module(line_monitor, 'subprocess') # this replaces the subprocess object␣
   ↪inside line_monitor with a Fake("subprocess") object
8      patch_module(line_monitor, 'pty') # does the same for the pty module
9
10 def test_lauch_subprocess_with_pseudoterminal(override_imports):
11     tested = line_monitor.LineMonitor()
12     with Scenario() as s:
13         s.pty.openpty() >> ('write_to_fd', 'read_from_fd')
14         s.subprocess.Popen(['my', 'command', 'line'], stdout='write_to_fd', close_
   ↪fds=True)
15
16         tested.launch_subprocess(['my', 'command', 'line'])
```

What's going on here?

- First, we use `patch_module` to mock imported modules `subprocess` and `pty`, as described above. Note that our test function depends on `override_imports` to make everything work.
- In our `Scenario` we demand two things:
  - That our code calls `pty.openpty()` to create a pseudo-terminal and obtain its two file descriptors.
  - That our code then launch a subprocess and point its `stdout` to the write file-descriptor of the pseudo-terminal (we also demand `close_fds=True` wince we want to fully specify our subprocess's inputs and outputs).
- Finally, we call our `.launch_subprocess()` method to actually do the work - we can't hope that our code meet our expectations if we never actually call it, right?

A few points on this:

- See how we *first* write our expectations and only *then* call the code to deliver on these expectations. This is one way Testix pushes you into a Test Driven mindset.

- In real life, `pty.openpty()` returns two file descriptors - which are *integers*. In our test, we made this call return two *strings*.

  We could have, e.g. define two constants equal to some integers, e.g. `WRITE_FD=20` and `READ_FD=30` and used those - but it wouldn't really matter and would make the test more cluttered. Technically, what's important is that `openpty()` returns a tuple and we demand that the first item in this tuple is passed over to the right place in the call to `Popen()`. Some people find fault with this style. Personally I think passing strings around (recall that in Python strings are immutable) where all you're testing is moving around objects - is a good way to make a readable test.

Failing the Test

Remember, when practicing TDD you should always fail your tests first, and make sure they *fail properly*.

So let's see some failures! Let's see some RED!

Running this test with the *skeleton implementation* we have for `LineMonitor` results in:

```
E       Failed:
E       testix: ScenarioException
E       testix details:
E       Scenario ended, but not all expectations were met. Pending expectations␣
↪(ordered): [pty.openpty(), subprocess.Popen(['my', 'command', 'line'], stdout = 'write_
↪to_fd', close_fds = True)]
```

Very good, our tests fails as it should: the test expects, e.g. `openpty()` to be called, but our current implementation doesn't call anything - so the test fails in disappointment.

Now that we have our RED, let's get to GREEN.

Passing the Test

Let's write some code that makes the test pass:

```
1  import subprocess
2  import pty
3
4  class LineMonitor:
5      def register_callback(self, callback):
6          pass
7
8      def launch_subprocess(self, *popen_args, **popen_kwargs):
9          write_to, read_from = pty.openpty()
10         popen_kwargs['stdout'] = write_to
11         popen_kwargs['close_fds'] = True
12         subprocess.Popen(*popen_args, **popen_kwargs)
13
14     def monitor(self):
15         pass
```

Running our test with this code produces

```
test_line_monitor.py::test_lauch_subprocess_with_pseudoterminal PASSED
```

Finally, we see some GREEN!

Usually we will now take the time to REFACTOR our code, but we have so little code at this time that we'll skip it for now.

OK, we have our basic subprocess with a pseudo-terminal - now's the time to test for and implement actually monitoring the output.

Monitoring The Output

Next we want to test the following behaviour: we register a callback with our `LineMonitor` object using its `.register_callback()` method, and it calls our callback with each line of output it reads from the pseudo-terminal.

Python streams have a useful `.readline()` method, so let's wrap the read file-descriptor of the pseudo-terminal with a stream. It turns out that you can wrap a file descriptor with a simple call to the built-on `open()` function, so we'll use that.

Note that we *add a new test*, leaving the previous one intact. This means that we *keep everything we already have working*, while we add a test for this new behaviour.

Let's start by describing a scenario where we read several lines from the pseudo-terminal and demand that they are transferred to our callback.

```

def test_receive_output_lines_via_callback(override_imports):
    tested = line_monitor.LineMonitor()
    with Scenario() as s:
        s.pty.openpty() >> ('write_to_fd', 'read_from_fd')
        s.open('read_from_fd', encoding='latin-1') >> Fake('reader') # wrapping a binary␣
↪file descriptor with a text-oriented stream requires an encoding
        s.subprocess.Popen(['my', 'command', 'line'], stdout='write_to_fd', close_
↪fds=True)

        tested.launch_subprocess(['my', 'command', 'line'])

        s.reader.readline() >> 'line 1'
        s.my_callback('line 1')
        s.reader.readline() >> 'line 2'
        s.my_callback('line 2')
        s.reader.readline() >> 'line 3'
```

What's going on here?

- We add a demand that our code create a Python stream from the pseudo-terminal's read-descriptor before launching the subprocess.
- We then call .launch_subprocess() to meet those demands.
- We describe the "read-from-pseudo-terminal-forwared-to-callback" data flow for 3 consecutive lines.
- We register a `Fake('my_callback')` object as our callback - this way, when the code calls the callback, it will be meeting our demands in this test. It's important that `'my_callback'` is used as this `Fake`'s name, since we refer to it in the `Scenario`.
- We then call the `.monitor()` method - this method should do all the reading and forwarding.

We must also remember to mock the built-in open:

```
@pytest.fixture
def override_imports(patch_module):
    patch_module(line_monitor, 'subprocess') # this replaces the subprocess object␣
↪inside line_monitor with a Fake("subprocess") object
    patch_module(line_monitor, 'pty') # does the same for the pty module
    patch_module(line_monitor, 'open')
```

We can already see a problem: the scenario is actually built out of two parts - the part which tests `.launch_subprocess()`, and the part which tests `.monitor()`.

Furthermore, since we have our previous test in `test_lauch_subprocess_with_pseudoterminal`, which doesn't expect the call to `open()`, the two tests are in contradiction.

The way to handle this is to refactor our test a bit:

```
from testix import *
import pytest
import line_monitor

@pytest.fixture
def override_imports(patch_module):
    patch_module(line_monitor, 'subprocess')
    patch_module(line_monitor, 'pty')
    patch_module(line_monitor, 'open')

def launch_scenario(s):
    s.pty.openpty() >> ('write_to_fd', 'read_from_fd')
    s.open('read_from_fd', encoding='latin-1') >> Fake('reader')
    s.subprocess.Popen(['my', 'command', 'line'], stdout='write_to_fd', close_fds=True)

def test_lauch_subprocess_with_pseudoterminal(override_imports):
    tested = line_monitor.LineMonitor()
    with Scenario() as s:
        launch_scenario(s)
        tested.launch_subprocess(['my', 'command', 'line'])

def test_receive_output_lines_via_callback(override_imports):
    tested = line_monitor.LineMonitor()
    with Scenario() as s:
        launch_scenario(s)
        tested.launch_subprocess(['my', 'command', 'line'])

        s.reader.readline() >> 'line 1'
        s.my_callback('line 1')
        s.reader.readline() >> 'line 2'
        s.my_callback('line 2')
        s.reader.readline() >> 'line 3'
        s.my_callback('line 3')

        tested.register_callback(Fake('my_callback'))
        tested.monitor()
```

By convention, helper functions that help us modify scenarios end with _scenario.

OK this seems reasonable, let's get some RED! Running this both our tests fail:

```
E       Failed:
E       testix: ExpectationException
E       testix details:
E       === Scenario (no title) ===
E        expected: open('read_from_fd', encoding = 'latin-1')
E        actual  : subprocess.Popen(['my', 'command', 'line'], stdout = 'write_to_fd',␣
↪close_fds = True)
```

We changed our expectations from `.launch_subprocess()` to call `open()`, but we did not change the implementation yet, so Testix is surprised to find that we actually call `subprocess.Popen` - and makes our test fail.

Good, let's fix it and get to GREEN. We introduce the following to our code:

```
import subprocess
import pty

class LineMonitor:
    def __init__(self):
        self._callback = None

    def register_callback(self, callback):
        self._callback = callback

    def launch_subprocess(self, *popen_args, **popen_kwargs):
        write_to, read_from = pty.openpty()
        popen_kwargs['stdout'] = write_to
        popen_kwargs['close_fds'] = True
        self._reader = open(read_from, encoding='latin-1')
        subprocess.Popen(*popen_args, **popen_kwargs)

    def monitor(self):
        for _ in range(3):
            line = self._reader.readline()
            self._callback(line)
```

This passes the test, but that's not really what we meant - right? Obviously we would like a `while True` to replace the for _ in range(3) here.

However, if we write a `while True`, then Testix will fail us for the 4th call to `.readline()`, since it only expects 3 calls.

Testing infinite, `while True` loops is a problem, but we can get around it by injecting an exception that will terminate the loop. Just as we can determine what calls to `Fake` objects return, we can make them raise exceptions.

Testix even comes with an exception class just for this use case, `TestixLoopBreaker`.` Let's introduce another ``.readline()` expectation into our test, using Testix's `Throwing` construct:

```
def test_receive_output_lines_via_callback(override_imports):
    tested = line_monitor.LineMonitor()
    with Scenario() as s:
        launch_scenario(s)
        tested.launch_subprocess(['my', 'command', 'line'])

        s.reader.readline() >> 'line 1'
        s.my_callback('line 1')
```

(continues on next page)

(continued from previous page)

```
        s.reader.readline() >> 'line 2'
        s.my_callback('line 2')
        s.reader.readline() >> 'line 3'
        s.my_callback('line 3')
        s.reader.readline() >> Throwing(TestixLoopBreaker) # this tells the Fake('reader
↪') to raise an instance of TestixLoopBreaker()

        tested.register_callback(Fake('my_callback'))
        with pytest.raises(TestixLoopBreaker):
            tested.monitor()
```

NOTE - we once more *change the test first*. Also note that we can use `Throwing` to raise any type of exception we want, not just `TestixLoopBreaker`.

This gets us back into the RED.

```
E           Failed: DID NOT RAISE <class 'testix.TestixLoopBreaker'>
```

Since our code calls `.readline()` 3 times exactly, the fourth call, which would have resulted in `TestixLoopBreaker` being raised, did not happen.

Let's fix our code:

```
    def monitor(self):
        while True:
            line = self._reader.readline()
            self._callback(line)
```

And we're back in GREEN.

Edge Case Test: When There is no Callback

What happens if `.monitor()` is called, but no callback has been registered? We can of course implement all kinds of behaviour, for example, we can make it "illegal", and raise an Exception from `.monitor()` in such a case.

However, let's do something else. Let's just define things such that output collected from the subprocess when no callback has been registered is discarded.

```
def test_monitoring_with_no_callback(override_imports):
    tested = line_monitor.LineMonitor()
    with Scenario() as s:
        launch_scenario(s)
        tested.launch_subprocess(['my', 'command', 'line'])

        s.reader.readline() >> 'line 1'
        s.reader.readline() >> 'line 2'
        s.reader.readline() >> 'line 3'
        s.reader.readline() >> Throwing(TestixLoopBreaker) # this tells the Fake('reader
↪') to raise an instance of TestixLoopBreaker()

        with pytest.raises(TestixLoopBreaker):
            tested.monitor()
```

Notice there's no `.register_callback()` here. We demand that `.readline()` be called, but we don't demand anything else.

Running this fails with a RED

```
    def monitor(self):
        while True:
            line = self._reader.readline()
>           self._callback(line)
E           TypeError: 'NoneType' object is not callable
```

Which reveals that we in fact, did not handle this edge case very well.

Let's add code that fixes this.

```
    def monitor(self):
        while True:
            line = self._reader.readline()
            if self._callback is None:
                continue
            self._callback(line)
```

Our test passes - back to GREEN.

Edge Case Test: Asynchronous Callback Registration

What happens if we start monitoring without a callback, wait a while, and only then register a callback?

This allows a use case where we call the `.monitor()` (which blocks) in one thread, and register a callback in another thread.

Let's decide that in this case, the callback will receive output which is captured only after the callback has been registered.

Our test will be similar to `test_receive_output_lines_via_callback()`, however, we need to somehow make `tested.register_callback()` happen somewhere between one `.readline()` and the next. This is not so easy to do because of the same `while True` that gave us some trouble before.

Testix allows us to simulate asynchronous events like this using its `Hook` construct. Essentially `Hook(function, *args, **kawrgs)` can be injected into the middle of a `Scenario`, and it will call `function(*args, **kwargs)` at the point in which it's injected.

Here's how to write such a test:

```
def test_callback_registered_mid_monitoring(override_imports):
    tested = line_monitor.LineMonitor()
    with Scenario() as s:
        launch_scenario(s)
        tested.launch_subprocess(['my', 'command', 'line'])

        s.reader.readline() >> 'line 1'
        s.reader.readline() >> 'line 2'
        s.reader.readline() >> 'line 3'
        s << Hook(tested.register_callback, Fake('my_callback')) # the hook will execute␣
↪right after the 'line 3' readline finishes
        s.my_callback('line 3') # callback is now registered, so it should be called
```

(continues on next page)

(continued from previous page)

```
        s.reader.readline() >> Throwing(TestixLoopBreaker)

        with pytest.raises(TestixLoopBreaker):
            tested.monitor()
```

When we run it, we discover it's already GREEN! Oh no!

Turns out our previous change already solved this problem. This happens sometimes in TDD, so to deal with it, we revert our previous change and make sure this test becomes RED - and carefully check that it failed properly. Happily, this is the case for this particular test.

Let's Recap

We now have our first implementation of the `LineMonitor`. It essentially works, but it's still has its problems. We'll tackle these problems later in this tutorial, but first, let's do a short recap.

Recap

Let's recap a bit on what we've been doing in this tutorial.

We started with a test for the basic subprocess-launch behaviour, got to RED, implemented the code, and got to GREEN.

Next, when moving to implement the actual output monitoring behaviour, we kept the first test, and added a new one. This is very important in TDD - the old test keeps the old behaviour intact - if, when implementing the new behaviour we break the old one - we will know.

When working with Testix, you are encourage to track all your mocks (`Fake` objects) very precisely. This effectively made us refactor the launch-process test scenario into a `launch_scenario()` helper function, since you must launch a subprocess before monitoring it.

We also saw that adding a call to `open` made the original launch-process test fail as well as the new monitor test. This makes sense, since the launching behaviour now includes a call to `open` that it didn't before - and the code doesn't support that yet, so the test fails.

Another thing we ran into is that sometimes we get GREEN even when we wanted RED. This should make you uneasy - it usually means that the test is not really testing what you think it is. In our case, however, it was just because an edge case which we added a test for was already covered by our existing code. When that happens, strict TDD isn't really possible - and you need to revert to making sure that if you break the code on purpose, it breaks the test in the proper manner.

YAGNI

Another thing to notice, is that the call to `Popen` is simply

```
subprocess.Popen(*popen_args, **popen_kwargs)
```

And not, for example,

```
self._process = subprocess.Popen(*popen_args, **popen_kwargs)
```

Why didn't we save the subprocess in an instance variable? Working TDD makes us want to get to GREEN - no more, no less. Since we don't need to store the subprocess to pass the test, we don't do it.

Let me repeat that for you: **if we don't need it to pass the test, we don't do it**.

You might say "but we need to hold on to the subprocess to control it, see if it's still alive, or kill it".

Well, maybe we do. If that's what we really think, we should express this need in a test - make sure it's RED, and then write the code to make it GREEN.

This is one particular way of implementing the YAGNI principle - if you're not familiar with it, you should take the time to read about it.

Upholding the YAGNI requires a special kind of discipline, and TDD and Testix in particular, helps us achieve it.

Code Recap

Before we continue, here is the current state of our unit test and code.

The test:

```
from testix import *
import pytest
import line_monitor

@pytest.fixture
def override_imports(patch_module):
    patch_module(line_monitor, 'subprocess')
    patch_module(line_monitor, 'pty')
    patch_module(line_monitor, 'open')

def launch_scenario(s):
    s.pty.openpty() >> ('write_to_fd', 'read_from_fd')
    s.open('read_from_fd', encoding='latin-1') >> Fake('reader')
    s.subprocess.Popen(['my', 'command', 'line'], stdout='write_to_fd', close_fds=True)

def test_lauch_subprocess_with_pseudoterminal(override_imports):
    tested = line_monitor.LineMonitor()
    with Scenario() as s:
        launch_scenario(s)
        tested.launch_subprocess(['my', 'command', 'line'])

def test_receive_output_lines_via_callback(override_imports):
    tested = line_monitor.LineMonitor()
    with Scenario() as s:
        launch_scenario(s)
        tested.launch_subprocess(['my', 'command', 'line'])

        s.reader.readline() >> 'line 1'
        s.my_callback('line 1')
        s.reader.readline() >> 'line 2'
        s.my_callback('line 2')
        s.reader.readline() >> 'line 3'
        s.my_callback('line 3')
        s.reader.readline() >> Throwing(TestixLoopBreaker) # this tells the Fake('reader
↪') to raise an instance of TestixLoopBreaker()

        tested.register_callback(Fake('my_callback'))
        with pytest.raises(TestixLoopBreaker):
            tested.monitor()
```

and the code that passes it

```
import subprocess
import pty

class LineMonitor:
    def __init__(self):
        self._callback = None

    def register_callback(self, callback):
        self._callback = callback

    def launch_subprocess(self, *popen_args, **popen_kwargs):
        write_to, read_from = pty.openpty()
        popen_kwargs['stdout'] = write_to
        popen_kwargs['close_fds'] = True
        self._reader = open(read_from, encoding='latin-1')
        subprocess.Popen(*popen_args, **popen_kwargs)

    def monitor(self):
        while True:
            line = self._reader.readline()
            self._callback(line)
```

Better Monitoring

If you try working with our current `LineMonitor` implementation you will find it has some disadvantages.

- There is no way to stop monitoring.
- In particular, if the underlying subprocess crashes, the monitor will just block for ever - it is blocked trying to `.readline()` - but the line will never come.

Furthermore, we originally wanted the ability to have more than one callback.

Let's improve our `LineMonitor`, starting by handling the underlying subprocess a little more carefully.

TBD

4.2 Reference

4.2.1 Context Manager Support

TODO

4.2.2 AsyncIO Support

TODO

4.2.3 Less Exact Expectations

TODO