Testix

Release 11.0.0

Yoav Kleinberger

Aug 23, 2023

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:

1. setup mock
2. let code do something with mock
3. assert mock used in correct way

Testix flow is a bit different

1. setup mock objects
2. 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:

1. setup our expectations (these are the `s.sock.*` lines)
2. enforce our expectations (this is done by the `with` statement)

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

CHAPTER

THREE

SOME ADVANCED FEATURES

Testix natively and elegantly supports testing for

1. Context managers (`with` statement constructs)
2. `async` code
3. `async` context managers (`async with` statement constructs)
4. Hooks - allowing you to simulate asynchronous events that happen between two lines of your code

CHAPTER

FOUR

ADVANTAGES

Testix has been written to promote the following

1. 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)`
2. 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?

Go to the *reference* or read the *Tutorial*

CHAPTER

FIVE

READ MORE

5.1 Reference

5.1.1 Basic Usage: Scenarios and Fake Objects

Scenarios and Fake Objects

Testix is a mocking framework designed to support the TDD style of programming. When writing a test with Testix, we use a Scenario() object to specify *what should happen* or what we *expect* should happen when the tested code is run. We can refer to these as *demands* or *expectations*.

Here's a test that expects the tested code to repeatedly call `.recv(4096)` on a socket, until an empty sequence is returned, and then call `.close()` on the socket. Furthermore, `tested.read()` should return the accumulated data.

```
from testix import *

import reader

def test_read_all_from_socket():
    with Scenario() as s:
        s.sock.recv(4096) >> b'data1'
        s.sock.recv(4096) >> b'data2'
        s.sock.recv(4096) >> b'data3'
        NO_MORE_DATA = b''
        s.sock.recv(4096) >> b''
        s.sock.close()

        tested = reader.Reader(Fake('sock'))
        assert tested.read() == b'data1data2data3'
```

Scenarios track fake objects - instances of `Fake`. Fake objects have a name, e.g. `Fake('sock')` - and you can demand various method calls on a fake object, e.g.

```
s.sock.recv(4096) >> b'data1'
```

Means "we require that the `.recv()` method be called on the fake object named `'sock'` with exactly one argument: the number 4096. When this happens, this function call will return `b'data1'` to the caller".

Note this last part - when we define an *expectation*, we may also define the return value returned should the expectation come true.

The code which passes this test is

```
class Reader:
    def __init__(self, socket):
        self._sock = socket

    def read(self):
        accumulated = b''
        while True:
            data = self._sock.recv(4096)
            if data == b'':
                self._sock.close()
                return accumulated
            accumulated += data
```

The `Scenario()` object helps us define our expectations from our code, and also *enforces* these expectations when used, as above, in a `with Scenario() as s` statement. When the `with` ends, the `Scenario` object will make sure that *all expectations have been exactly met*.

Try to comment out some lines in the `Reader` class, and you will see that the test no longer passes. E.g. if you comment out the `.close()` call you will get

```
>       return pytest.fail( message )
...
E       Scenario ended, but not all expectations were met. Pending expectations␣
↪(ordered): [sock.close()]
```

Testix tells you that not all expectations were met, like it should.

More Complex Expectations

You can specify any number of arguments, and also keyword arguments, e.g.

```
s.alpha.func1(1, 2, a=1, b='hi there')
```

This will ensure that the `.func1()` method is called on the `Fake("alpha")` object *exactly* like this:

```
def my_code(a):
    a.func1(1, 2, a=1, b='hi there')
```

With this definition, `my_code(Fake("alpha"))` will pass the test. The value returned from `.func1()` in this case will be `None`. If you want to specify a return value, use >> as before

```
s.alpha.func1(1, 2, a=1, b='hi there')
```

Overriding Imported Modules With Fake Objects

Since `Scenario` can only track `Fake` objects, the tested code must have access to them. We already saw one way this can happen, when we pass in a fake, e.g.

```
tested = reader.Reader(Fake('sock'))
```

Another common pattern with Testix is to override some global names inside the tested module - this essentially overrides `import` statements.

Here is an example of overriding the `socket` import using Testix's `patch_module` pytest fixture. This test demands that the `MyServer()` object create a socket, listen on it, accept a connection, send `b'hi'` over this connection, then close it.

```
from testix import *
import pytest
import my_server

@pytest.fixture(autouse=True) # if autouse is not used here, you will have to specify␣
↪override_imports as an argument to test_my_server() below
def override_imports(patch_module):
    patch_module(my_server, 'socket') # replace socket with Fake('socket')

def test_my_server():
    with Scenario() as s:
        s.socket.socket() >> Fake('server_sock')
        s.server_sock.listen()

        s.server_sock.accept() >> (Fake('connection'), 'some address info')
        s.connection.send(b'hi')
        s.connection.close()

        tested = my_server.MyServer()
        tested.serve_request()

```

NOTE: The `patch_module` helper will, when the test is over, return the original object to its place. It's important to use `patch_module` and not do it yourself.

Another important point is that `patch_module` overrides global names, go, e.g. if we use `patch_module` like this

```
patch_module(my_module, 'xxx')
```

And `my_module` has this code

```
xxx = 300

def get_xxx():
    return xxx
```

Then `xxx` will not be 300 for the duration of the test, but instead have a fake object by the same name `Fake("xxx")`.

This will also be the case if `my_module` had

```
from important_constants import xxx

def get_xxx():
    return xxx
```

Using `patch_module` To Mock Builtin Objects

We can also use `patch_module` to override builtin objects, such as the function `open()`.

```
patch_module(my_module, 'open')

# sometime later
s.open('some_file.txt', 'w') >> Fake('open_file')
```

Using `patch_module` With Arbitrary Values

Usually we use `patch_module` to override module-level names with Fake objects, but you can specify any object as the override

```
patch_module(my_module, 'xxx', 500) # override xxx value with 500
```

The Most Common Ways To Create Fake Objects

So, Fake objects can be created and passed in directly, they can be used to mock imported modules using `patch_module`, and they can be returned as the result of another Fake object call, e.g. the line

```
s.socket.socket() >> Fake('server_sock')
```

In fact, in this line as well as in the one that follows:

```
s.server_sock.accept() >> (Fake('connection'), 'some address info')
```

There is, in fact, another method that creates Fake objects. The preceding line specifies that `server_sock.accept` is called - which, under the hood, implies the creation of a `Fake("server_sock.accept")` fake object.

To summarize, the main modes where fakes are created are:

1. Created and passed in directly
2. Created and returned as the return value of an expectation
3. Replacing a global name (usually an imported module) using `patch_module`
4. Implicitly created when addressing a method of a fake object, e.g. `server_sock.accept` above.

5.1.2 Less Strict Expectations

Less Specific Arguments

Sometimes you want to specify an expectation, but with less strict demands.

Continuing with the previous example, maybe we don't care that much that `4096` be used when calling `.recv()`

This can be accomplished using `IgnoreArgument()`, e.g.

```
from testix import *

import reader

```

(continues on next page)

(continued from previous page)

```
def test_read_all_from_socket():
    with Scenario() as s:
        s.sock.recv(IgnoreArgument()) >> b'data1'
        s.sock.recv(IgnoreArgument()) >> b'data2'
        s.sock.recv(IgnoreArgument()) >> b'data3'
        NO_MORE_DATA = b''
        s.sock.recv(IgnoreArgument()) >> b''
        s.sock.close()

        tested = reader.Reader(Fake('sock'))
        assert tested.read() == b'data1data2data3'
```

You can also use `IgnoreArgument()` to specify that you demand some kwarg be used, but you don't care about it's value

```
s.alpha.func1(1, 2, a=IgnoreArgument(), b='hi there')

# must use (1, 2, a=<any object here>, b='hi there')
```

Unordered Expectations

Most of the time, in my experience, it's a good idea that expectations are met in the exact order that they were specified.

```
s.alpha.func1(1, 2)
s.alpha.func2('this must come after func1')
```

These expectations will only be met if `.func2()` is called after `.func1()` was called.

However, sometimes we want to relax this a little, and demand that some function is called, but you don't care if it's before or after some other function. You can do this by using the `.unordered()` modifier:

```
from testix import *

def test_unordered_expecation():
    with Scenario() as s:
        s.some_object('a')
        s.some_object('b')
        s.some_object('c').unordered()

        my_fake = Fake('some_object')
        my_code(my_fake)

def my_code(x):
    x('a')
    x('c')
    x('b')
```

This demands that the fake is called with `'b'` only *after* it was called with `'a'` - but it forgives the call with `'c'` - you can call the fake with `'c'` before, after on in between the `'a'` and `'b'` calls.

The code in `my_code()` passes this test.

Of course you can still use the >> operator to specify return values for your expectation.

Everlasting Expectations

Sometimes we don't want to test for a specific number of calls, but we do want to make sure that a function call is of a particular form.

We can modify an `.unordered()` expectation with `.everlasting()` and this essentially modifies the expectation to mean "this call is expected zero or more times, in any order".

Here's a small example

```
from testix import *

def test_unordered_expecation():
    with Scenario() as s:
        s.some_object('a')
        s.some_object('b')
        s.some_object('c').unordered().everlasting()

        my_fake = Fake('some_object')
        my_code(my_fake)

def my_code(x):
    x('c')
    x('c')
    x('c')

    x('a')
    x('c')
    x('b')

    x('c')
    x('c')
```

Of course you can still use the >> operator to specify return values for your expectation.

5.1.3 Context Manager Expectations

Sometimes we want to demand that an object is used as a context manager in a `with` statement.

Here's an example of how to demand this on a fake object named `'locker'`, using `Scenarios` special `__with__` modifier, along with the code that passes the test.

```
from testix import *

def test_fake_context():
    locker_mock = Fake('locker')
    with Scenario() as s:
        s.__with__.locker.Lock() >> Fake('locked')
        s.locked.read() >> 'value'
        s.locked.updater.go('another_value')
        my_code(locker_mock)


def my_code(locker):
```

(continues on next page)

(continued from previous page)

```
    with locker.Lock() as locked:
        whatever = locked.read()
        locked.updater.go(f'another_{whatever}')
```

5.1.4 Advanced Argument Expectations

Most of our examples of working with `Scenario` expectations are of *exact* matches, e.g.

```
s.classroom.set('A', index=9, name='Alpha')
```

Means we expect the `.set()` method to be called on the fake object `Fake('classroom')` with these *exact* arguments: a positional argument with the value `'A'`, and two keyword arguments `index=9, name='Alpha'`.

In most cases this is exactly what we want. However, sometimes we want something else. Let's see what Testix supports.

Ignoring A Specific Argument

Sometimes we don't care about the value of a specific argument. For example, we might want to verify that our code calls `time.sleep()` but we don't want to test for a specific number of seconds.

Here's how to do this in Testix:

```
with Scenario() as s:
    s.time.sleep(IgnoreArgument())
    # must call time.sleep() with one argument, but any value for this argument will be␣
↪OK
```

You can also use this in a kwarg expectation

```
with Scenario() as s:
    s.greeter.greet('hello!', person=IgnoreArgument())
    # must use 'hello!', but any value for person will be OK
```

Ignoring All Call Details

What if we want to make sure a method is called, but we don't care about the arguments at all? This is what `IgnoreCallDetails()` is for, e.g. this test expects the `.connect()` method to be called on the fake object `Fake('database')` three times, but specifies that it doesn't care about the exact arguments:

```
from testix import *

import server

def test_person_connects_somehow():
    with Scenario() as s:
        s.database.connect(IgnoreCallDetails())
        s.database.connect(IgnoreCallDetails())
        s.database.connect(IgnoreCallDetails())

        tested = server.Server(Fake('database'))
```

(continues on next page)

(continued from previous page)

```
        tested.connect1()
        tested.connect2()
        tested.connect3()
```

Here is an example of code that passes this test

```
class Server:
    def __init__(self, database):
        self._database = database

    def connect1(self):
        self._database.connect()

    def connect2(self):
        self._database.connect(1, 2, 3, x='y')

    def connect3(self):
        self._database.connect(a='1', b='2', c='3')
```

As you can see, the `.connect()` method is called three times, but with different arguments each time, and this satisfies the `IgnoreCallDetails()` expectation.

Testing for Object Identity

Sometimes we want to ensure a method is called with a specific object. We are not satisfied with it being called with an *equal* object, we want the *same actual object*. That is, we are interested in testing `actual is expected` and not `actual == expected`.

We can do this with `ArgumentIs`. Here is an example:

```
import pytest
from testix import *

import classroom

def test_this_will_pass():
    joe = classroom.Person('Joe')
    with Scenario() as s:
        s.mylist.append(ArgumentIs(joe))

        tested = classroom.Classroom(Fake('mylist'))
        tested.enter_original(joe)

def test_this_will_fail():
    joe = classroom.Person('Joe')
    with Scenario() as s:
        s.mylist.append(ArgumentIs(joe))

        tested = classroom.Classroom(Fake('mylist'))
        tested.enter_copy(joe)
```

We list here two tests, both demand that the object passed to `mylist.append()` will be the actual object `joe` created at the start of the test. The test `test_this_will_fail()` is made to fail on purpose by using the wrong method on

the tested object, this is the code that passed (and fails) these tests:

```
class Person:
    def __init__(self, name):
        self.name = name

    def __eq__(self, other):
        return self.name == other.name

class Classroom:
    def __init__(self, people: list):
        self._people = people

    def enter_original(self, student):
        self._people.append(student)

    def enter_copy(self, student):
        self._people.append(Person(student.name))
```

If we didn't use `ArgumentIs` and just used

```
s.mylist.append(joe)
```

The `test_this_will_fail()` test would have passed, because `joe` is equal to the object passed to `.append()` as defined by the `__eq__` method. With `ArgumentIs`, however, you will get something like this failure message:

```
E       testix: ExpectationException
E       testix details:
E       === Scenario (no title) ===
E        expected: mylist.append(|IS <classroom.Person object at 0x7f5162c2c5e0>|)
E        actual  : mylist.append(<classroom.Person object at 0x7f5162c2c250>)
```

Capturing Arguments

Sometimes we don't want to *demand* anything about a method's arguments, but we do want to *capture* them. This is useful for when we want to simulate the triggering of an internal callback.

For example, suppose we have a class which implements some logic when the process ends via an atexit <https://docs.python.org/3/library/atexit.html>_ handler. Testing this might seem hard, since we don't want to actually make the process (which is running our test) exit.

Here's how to do it using Testix's `SaveArgument` feature.

```
from testix import *
import pytest
import robot

@pytest.fixture
def mock_imports(patch_module):
    patch_module(robot, 'atexit') # mock atexit module

def test_atexit_handler(mock_imports):
    with Scenario() as s:
        s.atexit.register(saveargument.SaveArgument('the_handler'))
```

(continues on next page)

(continued from previous page)

```
        tested = robot.Robot(cleanup_func=Fake('cleanup_logic'))

        handler = saveargument.saved()['the_handler']
        s.cleanup_logic(1, 2, 3)

        handler()
```

We use `saveargument.SaveArgument()` to capture the argument passed to `atexit.register()`, and name this captured argument `the_handler`.

We later retrieve the captured callback via the `saveargument.saved()` dictionary.

This enables us to trigger the callback ourselves by calling `handler()` - which satisfies our demand `s.cleanup_logic(1, 2, 3)`.

NOTE a simpler way might have been to make the `_cleanup()` function public, and then we could just call it: `tested.cleanup()`. However, if this is not called in our code, we should not make it public, and since we are forbidden by good ethics from accessing a private function from outside the class, we need to capture it.

Implementing Arbitrary Argument Matching

Sometimes you need some complicated logic that Testix doesn't support out of the box.

You can define your own argument expectation classes with some arbitrary logic, and use them in your tests, by implementing classes derived from the `ArgumentExpectation` base class, which is essentially an interface:

```
class ArgumentExpectation:
    def __init__(self, value):
        self.expectedValue = value

    def ok(self, value):
        # returns true if value meets the expectation, false otherwise
        raise Exception("must override this")
```

The following tests implements a new `StartsWith` expectation, which expects a string that starts with a given prefix.

```
import pytest
from testix import *

import temporary_storage

@pytest.fixture(autouse=True)
def mock_builtin(patch_module):
    patch_module(temporary_storage, 'open')

class StartsWith(ArgumentExpectation):
    def ok(self, value):
        return value.startswith(self.expectedValue)

def test_person_connects_somehow():
    with Scenario() as s:
        s.open(StartsWith('/tmp/'), 'w') >> Fake('the_file')
        s.the_file.write('file_name: some_file\n\n')
```

(continues on next page)

(continued from previous page)

```
        tested = temporary_storage.TemporaryStorage()
        the_file = tested.create_file('some_file')
        assert the_file is Fake('the_file')
```

We demand that `open()` be called with a filename that starts with `/tmp/`, and with *exactly* `'w'` as the second argument.

This code passes this test:

```
class TemporaryStorage:
    def create_file(self, filename):
        f = open('/tmp/' + filename, 'w')
        f.write('file_name: ' + filename + '\n\n')
        return f
```

The `ArgumentExpectation` base class implements a default `__repr__` function, but you can implement one yourself e.g.

```
class StartsWith(ArgumentExpectation):
    def ok(self, value):
        return value.startswith(self.expectedValue)

    def __repr__(self):
        return 'StartsWith({})'.format(repr(self.expectedValue))
```

Using this `ArgumentExpectation` interface, you can make Testix support any arbitrary and complicated argument verification you need.

5.1.5 Hooks

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. In this example, we have a `LineProcessor` class that reads lines from a stream. If you register a callback with the `LineProcessor`, it will call it every time it reads a line.

We want to simulate the situation where a callback is registered (from another thread) in the middle of reading a stream, so we expect lines that are read after the callback is registered to be passed on to it.

```
with Scenario() as s:
    s.input_stream.readline() >> 'line 1'
    s.input_stream.readline() >> 'line 2'
    s << Hook(tested.register_callback, Fake('my_callback'))
    # the hook will execute right after the 'line 2' readline finishes
    # we therefore expect that after reading the next line, the callback will be called
    s.input_stream.readline() >> 'line 3'
    s.my_callback('line 3')
    s.input_stream.readline() >> 'line 4'
    s.my_callback('line 4')

    tested = LineProcessor(Fake('input_stream')) # no callbacks at this time
    tested.read_lines(Fake('input_stream'))
```

5.1.6 AsyncIO Support

Testix offers support for testing asynchronous code, that is code which takes advantage of Python's async and await keywords.

Testix's async support has been tested to work with pytest-asyncio.

AsyncIO Expectations

You can specify that a method call should be awaited using the `__await_on__` modifier of the `Scenario`, as in the example below. As you can see, you may also mix and match sync and async expectations.

```
from testix import *
import pytest

@pytest.mark.asyncio
async def test_async_expectations():
    with scenario.Scenario('awaitable test') as s:
        s.__await_on__.my_fake('some data') >> fake.Fake('another_fake')
        s.__await_on__.another_fake() >> fake.Fake('yet_another')
        s.__await_on__.yet_another() >> Fake('last_one')
        s.last_one.sync_func(1, 2, 3) >> 'sync value'

        assert await my_code(Fake('my_fake')) == 'sync value'

async def my_code(thing):
        another = await thing('some data')
        yet_another = await another()
        last_one = await yet_another()
        return last_one.sync_func(1, 2, 3)
```

Note that the test function itself is async and that you have to use the `pytest.mark.asyncio` decorator on the test - this decorator makes sure the test runs inside an `asyncio` event loop.

Note that the `__await_on__` changes the expectation `.my_fake('some data')` into *two* expectations - the function call, and the use of `await`-ing. You can see this, if, e.g., you cut `my_code(thing)` short by replacing its first line with `return 'sync value'`. This is the correct value, so the `assert` statement passes, however the `Scenario` context will inform you that

```
E       Scenario ended, but not all expectations were met. Pending expectations␣
↪(ordered):
[my_fake('some data'), await on my_fake('some data')@cb28287a42fc(),
another_fake(), await on another_fake()@94bb8afd7e45(),
yet_another(), await on yet_another()@b09a73bf3ee0(),
last_one.sync_func(1, 2, 3)]
```

You can see that every `__await_on__` results in a special expectation representing it.

AsyncIO Context Managers

You can specify your expectation for an object to be used as an async context manager (i.e. in an async with statement) by using the `__async_with__` modifier. Here's an example testing a module called `async_read` which has an async function `go()` which reads the contents of a file asynchronously.

```
from testix import *
import pytest

import async_read

@pytest.fixture(autouse=True)
def override_import(patch_module):
    patch_module(async_read, 'aiofiles')

@pytest.mark.asyncio
async def test_read_write_from_async_file():
    with scenario.Scenario() as s:
        s.__async_with__.aiofiles.open('file_name.txt') >> Fake('the_file')
        s.__await_on__.the_file.read() >> 'the text'

        assert 'the text' == await async_read.go('file_name.txt')
```

Note our use of `patch_module` to mock the aiofiles library, which we assume is imported and used by our `async_read` module.

The code which passes this test is

```
import aiofiles

async def go(filename):
    async with aiofiles.open(filename) as f:
        return await f.read()
```

Note you do not have to specify a return value with >> for the `__async_with__` expectation if you want to use the "anonymous" form of the `async with` statement:

```
async with lock(): # no "as" part
    await handle_critical_data()
```

5.2 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`.

5.2.1 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:

1. will launch subprocesses using `subprocess` under the hood
2. will allow the caller to register callbacks that get called from every line of output from the subprocess
3. 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

1. test logic
2. do not perform I/O (perhaps only to local files)
3. 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

1. perform some actual I/O
2. 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

1. an actual deployment, which is as similar as possible to real life deployments.
2. 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.

5.2.2 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}\n' 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*.

5.2.3 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}\n' 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

```

(continues on next page)

(continued from previous page)

```
8       def monitor(self):
9           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**,

1. we make sure they actually test what they pretend to to
2. we think hard about how to test this functionality - we gain focus on what our software is supposed to do
3. 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.

5.2.4 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

1. don't *really* want to send data over a *real* socket
2. 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

```
from testix import *

import data_sender

def test_sending_data():
    fake_socket = Fake('sock')
    with Scenario() as s:
        s.sock.send(b'the data')

        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 ')
9          s.sock.send(b'the data')
10
11         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

```
1
2  def send_some_data(socket, data):
3      length = len(data)
4      header = b'SIZE:' + bytes(str(length), encoding='latin-1')
5      socket.send(header)
6      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

```
1
2
3  def send_some_data(socket, data):
4      length = len(data)
5      header = b'SIZE:' + bytes(str(length), encoding='latin-1') + b' '
6      socket.send(header)
7      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)
        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 ')
```

(continues on next page)

(continued from previous page)

```
        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)
```

(continues on next page)

(continued from previous page)

```
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

1. the order in which calls are made
2. the exact arguments, positional and keyword, which are used
3. unexpected calls are considered a failure

Wrong Arguments

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

```
from testix import *
import my_code

def test_my_code():
    with Scenario() as s:
        s.source.get_names('all', order='lexicographic', ascending=True) >> ['some',
↪'names']
        s.destination.put_names(['some', 'names'])

        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:

1. use `Fake` objects to simulate various entities
2. use a `Scenario` object to define an exact set of demands or *expectations*
3. the `Scenario` object not only defines our expectations, it is also used in a `with` statement to *enforce* them.
4. 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.

1. 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.
2. 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.
3. Whatever expectations you define for you mock objects - they will be *exactly enforced* - defining expectations and enforcing them *is one and the same*.

5.2.5 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:

1. RED: write a *properly failing test*
2. GREEN: write code that passes the test - the code doesn't have to be pretty
3. 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:

1. a `LineMonitor` sill launch the subprocess using the subprocess Python standard library.
2. 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.

1. 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).
2. 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`.

```
from testix import *
import pytest
import line_monitor

@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
```

(continues on next page)

(continued from previous page)

```
    patch_module(line_monitor, 'pty') # does the same for the pty module

def test_lauch_subprocess_with_pseudoterminal(override_imports):
    tested = line_monitor.LineMonitor()
    with Scenario() as s:
        s.pty.openpty() >> ('write_to_fd', 'read_from_fd')
        s.subprocess.Popen(['my', 'command', 'line'], stdout='write_to_fd', close_
↪fds=True)

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

What's going on here?

1. 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.
2. 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).
3. 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:

1. 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.
2. 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:

```
import subprocess
import pty

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

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

    def monitor(self):
        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()
```

(continues on next page)

(continued from previous page)

```
    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?

1. We add a demand that our code create a Python stream from the pseudo-terminal's read-descriptor before launching the subprocess.
2. We then call .launch_subprocess() to meet those demands.
3. We describe the "read-from-pseudo-terminal-forwared-to-callback" data flow for 3 consecutive lines.
4. 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`.
5. 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')
```

(continues on next page)

(continued from previous page)

```

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):
```

(continues on next page)

(continued from previous page)

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

        tested.register_callback(Fake('my_callback'))
        with pytest.raises(loop_breaker.LoopBreaker):
            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 `LoopBreaker`.

This gets us back into the RED.

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

Since our code calls `.readline()` 3 times exactly, the fourth call, which would have resulted in `LoopBreaker` 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(loop_breaker.LoopBreaker) # this tells the Fake(
↪'reader') to raise an instance of TestixLoopBreaker()

        with pytest.raises(loop_breaker.LoopBreaker):
            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
        s.reader.readline() >> Throwing(loop_breaker.LoopBreaker)

        with pytest.raises(loop_breaker.LoopBreaker):
            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(loop_breaker.LoopBreaker) # this tells the Fake(
↪'reader') to raise an instance of TestixLoopBreaker()

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

and the code that passes it

```
import subprocess
import pty

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

(continues on next page)

(continued from previous page)

```

    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)
```

Watching The Subprocess

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

1. There is no way to stop monitoring.
2. In particular, if the underlying subprocess crashes, the monitor will just block forever - 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. We'll start by checking for available data before we try to read it.

Polling the Read File Descriptor

We want to create a polling object, and register the reader's file descriptor using its .register method.

Let's test for it. We have to mock the select module, of course, and also change our launch_scenario().

```
@pytest.fixture
def override_imports(patch_module):
    patch_module(line_monitor, 'subprocess')
    patch_module(line_monitor, 'pty')
    patch_module(line_monitor, 'open')
    patch_module(line_monitor, 'select')
    Fake('select').POLLIN = select.POLLIN

def launch_scenario(s):
    s.pty.openpty() >> ('write_to_fd', 'read_from_fd')
    s.open('read_from_fd', encoding='latin-1') >> Fake('reader')
    s.select.poll() >> Fake('poller')
    s.reader.fileno() >> 'reader_descriptor'
    s.poller.register('reader_descriptor', select.POLLIN)
    s.subprocess.Popen(['my', 'command', 'line'], stdout='write_to_fd', close_fds=True) >
↪> Fake('the_process')
```

There is a quick here - after running `patch_module(line_monitor, 'select')`, the `select` object inside the tested `line_monitor` module is replace by a `Fake('select')` fake object. Later, we want to demand that `poller.register()` be called with the `select.POLLIN` constant. As things are, this would technically also be the fake object `Fake('select.POLLIN')`, since Testix automatically generates fake objects whenever you lookup a `Fake`'s attribute (unless it's explicitly set up).

While it is possible to demand

```
s.poller.register('reader_descriptor', Fake('select').POLLIN)
```

And it will work just fine, I find it less readable. Therefore I'd rather "rescue" the `POLLIN` object from the real `select` and assign it to the fake `select`.

You may notice another quirk - the function `.fileno()` returns a file descriptor, which is an integer. However, in our test we make it return a string value, `'reader_descriptor'`, and later test that this value is transmitted to the `.register()` call on the polling object.

Of course it is possible to write something like

```
FAKE_FILE_DESCRIPTOR = 12121212
s.reader.fileno() >> FAKE_FILE_DESCRIPTOR
s.poller.register(FAKE_FILE_DESCRIPTOR, select.POLLIN)
```

This is totally legitimate. However, In my opinion, when testing the logic of "this object from *here* should get *there*", using strings (which are immutable in Python) may be more readable than using the correct data type.

Changing `launch_scenario` has changed our tests, let's run them, see if they fail:

```
$ python -m pytest -sv docs/line_monitor/tests/unit/11/test_line_monitor.py

...

E       Failed:
E       testix: ExpectationException
E       testix details:
E       === Scenario (no title) ===
E        expected: select.poll()
E        actual  : subprocess.Popen(['my', 'command', 'line'], stdout = 'write_to_fd',␣
↪close_fds = True)
```

Yay :) we have RED. Our tests expect the new `.poll()` logic, but our code, of course, is still not up to date. Of course, all of our tests now fail, since they all depend on `launch_scenario()` being followed exactly.

Let's get to GREEN with this and then continue with testing the actual polling:

```
1  import subprocess
2  import pty
3  import select
4
5  class LineMonitor:
6      def __init__(self):
7          self._callback = None
8
9      def register_callback(self, callback):
10         self._callback = callback
11
```

(continues on next page)

(continued from previous page)

```
    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')
        self._poller = select.poll()
        self._poller.register(self._reader.fileno(), select.POLLIN)
        subprocess.Popen(*popen_args, **popen_kwargs)

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

Now our tests pass once again. We have GREEN, but we haven't really added the actual feature we want to develop. We want the monitor to stop monitoring once the underlying subprocess is dead, and not get blocked trying to read a line that will never come.

This will involve using the poll object to poll the read descriptor to see that there's some data to read before calling .readline(). Since our tests already involve various scenarios calling .readline() - doing this TDD doesn't mean writing new tests - it means modifying the tests that we have.

This happens sometimes in TDD, and it's perfectly normal. Now, let's get to RED.

Looking at an excerpt from our tests:

```
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')
```

We want to demand that every `.readline()` is preceded by a `.poll()`, and to only be performed if there's input available. The `.poll()` call returns a list of `[(file_descriptor, events), ...]` pairs, where events is a bitmask of flags indicating the state of the file descriptor (e.g. `POLLIN | POLLOUT`).

Still, the sequence of `.poll()` and `.readline()` is sort-of "the new readline", it makes up a logical scenario, so let's write it as a scenario function, `read_line_scenario`.

Here is our `test_receive_output_lines_via_callback`, adapted to the new situation.

```

def read_line_scenario(s, line):
    s.poller.poll() >> [('reader_descriptor', select.POLLIN)]
    s.reader.readline() >> line

def end_test_scenario(s):
    s.poller.poll() >> Throwing(loop_breaker.LoopBreaker)
```

(continues on next page)

(continued from previous page)

```

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'])

        read_line_scenario(s, 'line 1')
        s.my_callback('line 1')
        read_line_scenario(s, 'line 2')
        s.my_callback('line 2')
        read_line_scenario(s, 'line 3')
        s.my_callback('line 3')
        end_test_scenario(s)

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

```
running this, we get |RED|

E       testix: ExpectationException
E       testix details:
E       === Scenario (no title) ===
E        expected: poller.poll()
E        actual  : reader.readline()
```

Very good. Now let's fix our code to pass the tests. Note that we did not yet add a test for the case where the file descriptor does not have any data to read - that come later. Always proceed in small, baby steps - and you'll be fine. Try to do it all at once, and you'll crash and burn.

Getting to GREEN is super easy, we add just this one line of code:

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

Well, this is GREEN, but adds little value. It's time for a serious test that makes sure that `.readline()` is called *if and only if* `POLLIN` is present. Let's get to RED.

We introduce a `skip_line_scenario()`, and introduce it into our existing tests, such that they represent the situation when sometimes there is no data to read.

```
from testix import *
import pytest
import line_monitor
import select

@pytest.fixture
```

(continues on next page)

(continued from previous page)

```
def override_imports(patch_module):
    patch_module(line_monitor, 'subprocess')
    patch_module(line_monitor, 'pty')
    patch_module(line_monitor, 'open')
    patch_module(line_monitor, 'select')
    Fake('select').POLLIN = select.POLLIN

def launch_scenario(s):
    s.pty.openpty() >> ('write_to_fd', 'read_from_fd')
    s.open('read_from_fd', encoding='latin-1') >> Fake('reader')
    s.select.poll() >> Fake('poller')
    s.reader.fileno() >> 'reader_descriptor'
    s.poller.register('reader_descriptor', select.POLLIN)
    s.subprocess.Popen(['my', 'command', 'line'], stdout='write_to_fd', close_fds=True) >
↪> Fake('the_process')

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 read_line_scenario(s, line):
    s.poller.poll() >> [('reader_descriptor', select.POLLIN)]
    s.reader.readline() >> line

def skip_line_scenario(s):
    s.poller.poll() >> [('reader_descriptor', 0)]

def end_test_scenario(s):
    s.poller.poll() >> Throwing(loop_breaker.LoopBreaker)

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'])

        read_line_scenario(s, 'line 1')
        s.my_callback('line 1')
        read_line_scenario(s, 'line 2')
        s.my_callback('line 2')
        read_line_scenario(s, 'line 3')
        s.my_callback('line 3')
        skip_line_scenario(s)
        skip_line_scenario(s)
        read_line_scenario(s, 'line 4')
        s.my_callback('line 4')
        end_test_scenario(s)

        tested.register_callback(Fake('my_callback'))
        with pytest.raises(loop_breaker.LoopBreaker):
```

(continues on next page)

(continued from previous page)

```
            tested.monitor()

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'])

        read_line_scenario(s, 'line 1')
        read_line_scenario(s, 'line 2')
        skip_line_scenario(s)
        read_line_scenario(s, 'line 3')
        skip_line_scenario(s)
        end_test_scenario(s)

        with pytest.raises(loop_breaker.LoopBreaker):
            tested.monitor()

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'])

        read_line_scenario(s, 'line 1')
        skip_line_scenario(s)
        read_line_scenario(s, 'line 2')
        read_line_scenario(s, '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
        end_test_scenario(s)

        with pytest.raises(loop_breaker.LoopBreaker):
            tested.monitor()
```

The idea here is simple - sometimes `.poll()` returns a result where the `POLLIN` flag is not set - and then we should skip the `.readline()`.

Do we have RED? Yes we do:

```
E       Failed:
E       testix: ExpectationException
E       testix details:
E       === Scenario (no title) ===
E        expected: poller.poll()
E        actual  : reader.readline()
```

Let's get to GREEN. This requires us to add the following to our code:

```
    def monitor(self):
        while True:
            poll_results = self._poller.poll()
```

(continues on next page)

(continued from previous page)

```
            _, events = poll_results[0]
            if not (events & select.POLLIN):
                continue
            line = self._reader.readline()
            if self._callback is None:
                continue
            self._callback(line)
```

This is GREEN but not the best code, the `.monitor()` function is becoming too long, time for the REFACTOR step in our RED-GREEN-REFACTOR loop.

```
    def monitor(self):
        while True:
            if not self._data_available_to_read():
                continue
            line = self._reader.readline()
            if self._callback is None:
                continue
            self._callback(line)

    def _data_available_to_read(self):
        poll_results = self._poller.poll()
        _, events = poll_results[0]
        return events & select.POLLIN
```

Ah, much nicer.

Solving the Blocking Problem

We are now in a position not to block forever when data does not arrive. To do that, we need to add a timeout on the `.poll` call - since as it is now, it may still block forever waiting for some event on the file.

Getting to RED is simple in principle, e.g. if we want a 10 seconds timeout, just change demands of our various scenarios, e.g.

```
def read_line_scenario(s, line):
    s.poller.poll(10) >> [('reader_descriptor', select.POLLIN)]
    # note the 10 second timeout above
    s.reader.readline() >> line

# similarly for other poll scenario functions
```

If we do this, however - and later on discover that a 60 second timeout is more reasonable, we will have to Test Drive the change from 10 to 60. This seems more annoying that it is helpful. Sometimes, tests can be *too* specific.

Testix has a way to specifically ignore the values of specific arguments - you specify the special value `IgnoreArgument()` instead of the overly specific `10`.

Here's how to use it in this case:

```
def read_line_scenario(s, line):
    s.poller.poll(IgnoreArgument()) >> [('reader_descriptor', select.POLLIN)]
    s.reader.readline() >> line
```

(continues on next page)

(continued from previous page)

```

def skip_line_scenario(s):
    s.poller.poll(IgnoreArgument()) >> [('reader_descriptor', 0)]

def end_test_scenario(s):
    s.poller.poll(IgnoreArgument()) >> Throwing(loop_breaker.LoopBreaker)
```

Using this we get to RED

```
E       testix: ExpectationException
E       testix details:
E       === Scenario (no title) ===
E        expected: poller.poll(|IGNORED|)
E        actual  : poller.poll()
```

Note the `|IGNORED|` annotation. Getting to green is now a matter of adding this timeout in our code:

```
    def _data_available_to_read(self):
        poll_results = self._poller.poll(10)
        _, events = poll_results[0]
        return events & select.POLLIN
```

And we have GREEN again.

Oops, a bug

If you try this code, you will find that there's a bug: in real life .poll() may return an empty list.

When we find a bug, the TDD way is of course to write a test that reproduces it, and then fix the code. In our case, let's add a `poll_returns_empty_scenario` and sprinkle it in our existing tests, thus covering the behaviour with and without a callback, etc.

```
def skip_line_on_empty_poll_scenario(s):
    s.poller.poll(IgnoreArgument()) >> []
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'])

        read_line_scenario(s, 'line 1')
        s.my_callback('line 1')
        read_line_scenario(s, 'line 2')
        s.my_callback('line 2')
        read_line_scenario(s, 'line 3')
        s.my_callback('line 3')
        skip_line_scenario(s)
        skip_line_scenario(s)
        skip_line_on_empty_poll_scenario(s)
        read_line_scenario(s, 'line 4')
        s.my_callback('line 4')
        end_test_scenario(s)
```

(continues on next page)

(continued from previous page)

```
21
22          tested.register_callback(Fake('my_callback'))
23          with pytest.raises(loop_breaker.LoopBreaker):
24              tested.monitor()
```

This gets us into RED territory

```
E       IndexError: list index out of range
```

Now let's fix the bug.

```
1       def _data_available_to_read(self):
2           poll_results = self._poller.poll(10)
3           if len(poll_results) == 0:
4               return False
5           _, events = poll_results[0]
6           return events & select.POLLIN
```

We are now GREEN, and, since we are working TDD, we have a test for this bug - *and it will not return in the future*.

Has the Subprocess Died?

We are now ready to add functionality to stop the monitor in case the subprocess itself has died. We will want our code to use `.poll()` on the `Popen` object itself, and if `.poll()` returns a non-None value, stop the monitor.

If you think about it, we can poll the subprocess only when there's no data available. It may be that there is data to read and process has died, but in that case, we'll just discover it is dead when the data runs out. This way we make sure we read all the data out of the pipe when the process has died, even if it takes more than one read.

If, however, there's no data to read, *and* the process is dead - then there's no point in continuing to monitor the pipe for more data, and we should close the reader, e.g.

```
1   def process_lives_scenario(s):
2       s.the_process.poll() >> None
3
4   def test_receive_output_lines_via_callback(override_imports):
5       tested = line_monitor.LineMonitor()
6       with Scenario() as s:
7           launch_scenario(s)
8           tested.launch_subprocess(['my', 'command', 'line'])
9
10          read_line_scenario(s, 'line 1')
11          s.my_callback('line 1')
12          read_line_scenario(s, 'line 2')
13          s.my_callback('line 2')
14          read_line_scenario(s, 'line 3')
15          s.my_callback('line 3')
16          skip_line_scenario(s)
17          process_lives_scenario(s)
18          skip_line_scenario(s)
19          process_lives_scenario(s)
20          skip_line_on_empty_poll_scenario(s)
21          process_lives_scenario(s)
```

(continues on next page)

(continued from previous page)

```
        read_line_scenario(s, 'line 4')
        s.my_callback('line 4')
        end_test_scenario(s)

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

Note that we demand process polling only after no data was ready to read, here it only comes after some `skip_line*scenario` function.

This brings us into RED territory. Current we have not taken the case that the process dies into account, but as usual, we're taking things slowly. Let's get to GREEN.

```
    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')
        self._poller = select.poll()
        self._poller.register(self._reader.fileno(), select.POLLIN)
        self._process = subprocess.Popen(*popen_args, **popen_kwargs)

    def monitor(self):
        while True:
            if not self._data_available_to_read():
                self._process.poll()
                continue
            line = self._reader.readline()
            if self._callback is None:
                continue
            self._callback(line)
```

Note that this is the first time we bothered to save the subprocess `Popen` object! This is another example of how TDD helps us. If the test passes without us making some move - then we simply don't make it. This helps us write minimalistic code. Remember, code that doesn't exist - has no bugs.

We are now in GREEN - so let's get into RED again, and add a specific test for the "process has died scenario":

```
def process_died_scenario(s):
    s.the_process.poll() >> 'some_exit_code'
    s.reader.close()

def test_receive_output_lines_via_callback__process_ends__orderly_close(override_
↪imports):
    tested = line_monitor.LineMonitor()
    with Scenario() as s:
        launch_scenario(s)
        tested.launch_subprocess(['my', 'command', 'line'])

        read_line_scenario(s, 'line 1')
        s.my_callback('line 1')
        read_line_scenario(s, 'line 2')
```

(continues on next page)

(continued from previous page)

```
        s.my_callback('line 2')
        read_line_scenario(s, 'line 3')
        s.my_callback('line 3')
        skip_line_scenario(s)
        process_lives_scenario(s)
        skip_line_scenario(s)
        process_died_scenario(s)

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

Note that we no longer need the `TestixLoopBreaker` trick - since we now expect the `.monitor()` function to simply finish and break out of its infinite loop.

Are we in RED? Yes we are:

```
E       testix: ExpectationException
E       testix details:
E       === Scenario (no title) ===
E        expected: reader.close()
E        actual  : poller.poll(10)
```

The test wants the infinite loop to finish and close the reader, but the code just goes on.

Let's fix our code:

```
    def monitor(self):
        while True:
            if not self._data_available_to_read():
                exit_code = self._process.poll()
                if exit_code is not None:
                    self._reader.close()
                    break
                continue
            line = self._reader.readline()
            if self._callback is None:
                continue
            self._callback(line)
```

We're GREEN, but this function has grown too long again, so let's REFACTOR.

```
    def monitor(self):
        while True:
            if not self._data_available_to_read():
                if self._alive():
                    continue
                self._cleanup()
                return
            line = self._reader.readline()
            if self._callback is None:
                continue
            self._callback(line)

    def _alive(self):
```

(continues on next page)

(continued from previous page)

```
        exit_code = self._process.poll()
        return exit_code is None

    def _cleanup(self):
        self._reader.close()

```

Not shorter, but more semantically clear, at least it my opinion. Since we have tests, we can refactor without fear - since we can always make sure we are still in the GREEN!

5.2.6 Conclusion

We can finally run our end-to-end test:

```
$ python -m pytest -sv docs/line_monitor/tests/e2e/test_line_monitor.py
============================================================================ test session starts␣
↪=============================================================================
platform linux -- Python 3.10.7, pytest-7.0.1, pluggy-1.0.0 -- /home/yoav/.cache/
↪pypoetry/virtualenvs/testix-rvJpiJ6N-py3.10/bin/python
cachedir: .pytest_cache
hypothesis profile 'default' -> database=DirectoryBasedExampleDatabase('/home/yoav/work/
↪testix/.hypothesis/examples')
rootdir: /home/yoav/work/testix
plugins: asyncio-0.16.0, cov-3.0.0, hypothesis-6.37.0
collected 1 item

docs/line_monitor/tests/e2e/test_line_monitor.py::test_line_monitor PASSED

============================================================================ 1 passed in 0.08s␣
↪=============================================================================
```

We have plenty of tests, and 100% code coverage.

```
docs/line_monitor/tests/unit/26/test_line_monitor.py::test_lauch_subprocess_with_
↪pseudoterminal PASSED
docs/line_monitor/tests/unit/26/test_line_monitor.py::test_receive_output_lines_via_
↪callback PASSED
docs/line_monitor/tests/unit/26/test_line_monitor.py::test_monitoring_with_no_callback␣
↪PASSED
docs/line_monitor/tests/unit/26/test_line_monitor.py::test_callback_registered_mid_
↪monitoring PASSED
docs/line_monitor/tests/unit/26/test_line_monitor.py::test_receive_output_lines_via_
↪callback__process_ends__orderly_close PASSED

---------- coverage: platform linux, python 3.10.7-final-0 -----------
Name                                          Stmts   Miss  Cover   Missing
---------------------------------------------------------------------------
docs/line_monitor/source/26/line_monitor.py      38      0   100%
```

Since we practiced Test Driven Development:

1. *Every single line* of our code is justified.

2. Every edge case we thought about is documented - via our tests. While tests written in Python are less readable than actual documentation written in English - they are much, much more reliable. If we have a good CI system to run our tests - this form of documentation does not get outdated.
3. Every bugfix that was performed was *first* reproduced with a test - and only *then* fixed in the code - *proving* that the bug is, in fact, fixed.
4. Our code is minimalist - we do not have unneeded code "just in case" which always ends up causing bugs.

But we should also mention the higher level benefits:

1. We coaxed the problem into an unambiguous, technically well-defined form - a bunch of tests.
2. Thus, we made sure that we understand the problem at hand.
3. Only then, did we proceed to try to solve it.
4. And when we did, we had the means to prove it.

Returning to the student analogy from "*The Importance of Proper Failure*", we gave the student a test to solve, instead of writing a test to fit the student's solution.

In one word - we were *logical* about it.

This is the proper way to write code. When done properly, it increases both development speed and the quality of the product delivered.

Now you know.

Use this knowledge. Write good code.

5.2.7 More About Readability

TDD is a great help for code quality and correctness.

However, there is more to high quality, readable code than TDD.

I've summarized good practices learned over many years, you are welcome to use them - see here.