BCG GAMMA

nibble

Advanced Testing Training

An introduction to Software testing
in a Machine Learning context

Agenda

I. Context and general presentation

II. Focus on End-to-End Testing

III. Focus on Integration Testing

IV. A few words about Performance Testing

V. Focus on Unit Testing — Day 3

VI. Code Refactoring principles

VII. Introduction to Continuous Integration

Part V — Focus on Unit Testing

- Details and facts about them
- Benefits
- Tools and techniques

Part V — Focus on Unit Testing

- **Details and facts about them**
- Benefits
- Tools and techniques

Narrowing down the scope…

White-box testing

Details about inner working are needed

… but don’t tie your test too close to implementation details.

Facts about unit tests

- Check the smallest parts of your code
- Force you to write cleaner code
  - Dependency injection
  - Separation of Concern
  - Single Responsibility Principle
- Good indicator of a bad implementation

They run fast!

You should run them as often as possible.

Part V — Focus on Unit Testing

- Details and facts about them
- **Benefits**
- Tools and techniques

Understanding the value might be challenging…

So, you're checking that "1 == 1" ?

Interesting...

Your code is made to evolve!

Benefits

- Easier debugging
- Easier refactoring
- Early targeting of code regressions

Part V — Focus on Unit Testing

- What are they
- Why do we use them
- **Tools and techniques**

Checking only the smallest parts, individually...

How is it even possible?

Useful tool #1

Mocks

Mocking

- Replace objects/variables by fake ones
- Configure them dynamically

```python
from unittest.mock import Mock

fake_model = LogisticRegression(...)
fake_model.predict = Mock(return_value=1)
```

```python
from unittest.mock import Mock

fake_model = LogisticRegression(...)
fake_model.predict = Mock(return_value=1)

prediction = fake_model.predict()
```

```python
from unittest.mock import Mock

fake_model = LogisticRegression(...)
fake_model.predict = Mock(return_value=1)

prediction = fake_model.predict()

assert prediction == 1
```

Mock also allows to make assertions on **how they have been used.**

https://docs.python.org/3/library/unittest.mock.html#unittest.mock.Mock.assert_called

```python
from unittest.mock import Mock

fake_model = LogisticRegression(...)
fake_model.predict = Mock(return_value=1)

prediction = fake_model.predict()

assert prediction == 1
fake_model.predict.assert_called_once()
```

`return_value` can be parametrized after declaration...

```python
from unittest.mock import Mock

fake_model = LogisticRegression(...)
fake_model.predict = Mock(return_value=1)

prediction = fake_model.predict()

assert prediction == 1
fake_model.predict.assert_called_once()
```

```python
from unittest.mock import Mock

fake_model = LogisticRegression(...)
fake_model.predict = Mock()

predict.return_value = 1
prediction = fake_model.predict()

assert prediction == 1
fake_model.predict.assert_called_once()
```

Mock **can trigger side effects** when they are called…

```python
from unittest.mock import Mock

fake_model = LogisticRegression(...)
fake_model.fit = Mock()
```

```python
from unittest.mock import Mock

fake_model = LogisticRegression(...)
fake_model.fit = Mock()

fake_model.fit.side_effect = RuntimeError(
    "Model fitting is forbidden."
)
```

```python
from unittest.mock import Mock

fake_model = LogisticRegression(...)
fake_model.fit = Mock()

fake_model.fit.side_effect = RuntimeError(
    "Model fitting is forbidden."
)
```

```python
from unittest.mock import Mock

fake_model = LogisticRegression(...)
fake_model.fit = Mock()

fake_model.fit.side_effect = RuntimeError(
    "Model fitting is forbidden."
)

fake_model.fit()
```

```python
from unittest.mock import Mock

fake_model = LogisticRegression(...)
fake_model.fit = Mock()

fake_model.fit.side_effect = RuntimeError(
    "Model fitting is forbidden."
)

fake_model.fit()
```

```
Traceback (most recent call last):
  File "tests/test_something.py", line 13, in <module>
    fake_model.fit()
  File "/usr/local/lib/python3.8/unittest/mock.py", line 1075, in __call__
    return self._mock_call(*args, **kwargs)
  File "/usr/local/lib/python3.8/unittest/mock.py", line 1079, in _mock_call
    return self._execute_mock_call(*args, **kwargs)
  File "/usr/local/lib/python3.8/unittest/mock.py", line 1134, in _execute_mock_call
    raise effect
RuntimeError: Model fitting is forbidden.
```

`side_effect` can be an **error to raise**, a **function to call**, or an **iterable**.

https://docs.python.org/3/library/unittest.mock.html#unittest.mock.Mock.side_effect

Useful tool #2

Monkey-patching

Monkey-patching

- Create mock classes or functions more easily, based on existing objects
- Update the logic of a function or class at **runtime**

```python
from unittest.mock import patch

# Either with a context manager…

with patch('src.my_function') as mock_function:
    assert isinstance(mock_function, Mock)
```

```python
from unittest.mock import patch

# Either with a context manager…

with patch('src.my_function') as mock_function:
    assert isinstance(mock_function, Mock)
```

Just pass the full Python path of the object you want to replace as first argument

```python
# Or with a decorator that will automatically
# provide the mock as a function argument

@patch('src.my_function')
def test_something(mock_function):
    assert isinstance(mock_function, Mock)
```

```python
# Or with a decorator that will automatically
# provide the mock as a function argument

@patch('src.my_function')
def test_something(mock_function):
    assert isinstance(mock_function, Mock)
```

```python
# Or with a decorator that will automatically
# provide the mock as a function argument

@patch('src.my_function')
def test_something(mock_function):
    assert isinstance(mock_function, Mock)
```

Use `autospec=True` to automatically create a mock **with the same spec than the object being replaced.**

```python
# src.py
def func(a, b, c):
    ...

# test.py
with patch('src.func', autospec=True) as mock_func:
    mock_func('Hello')
```

```python
# src.py
def func(a, b, c):
    ...

# test.py
with patch('src.func', autospec=True) as mock_func:
    mock_func('Hello')
```

```
Traceback (most recent call last):
  ...
TypeError: <lambda>() takes exactly 3 arguments (1 given)
```

Difficulty with patching:

Where to patch from ?

Rule of thumb

Patch from **where the object is looked up**, not from where it's defined.

https://docs.python.org/3/library/unittest.mock.html#where-to-patch

Example #1

```python
# module_a.py
def func():
    ...

# module_b.py
from module_a import func

def other():
    obj = func()
    ...
```

```python
# module_a.py
def func():
    ...

# module_b.py
from module_a import func

def other():
    obj = func()
    ...
```

Scenario

You want to test `module_b.other()` by patching `func`.

Where should your patch point to?

```
patch('module_a.func')

# ?
```

```python
# test.py
from module_b import other


@patch('module_a.func') ✖ No!
def test_other(mock_func):
    result = other()
    assert ...
```

```python
# test.py
from module_b import other


@patch('module_b.func') ✓ Yes!
def test_other(mock_func):
    result = other()
    assert ...
```

Example #2

```python
# module_a.py
def func():
    ...

# module_b.py
import module_a

def other():
    obj = module_a.func()
    ...
```

```python
# module_a.py
def func():
    ...

# module_b.py
import module_a

def other():
    obj = module_a.func()
    ...
```

Same scenario...

With a subtle difference!

```python
from module_b import other


@patch('module_b.func') ✖ No!
def test_other(mock_func):
    result = other()
    assert ...
```

```python
from module_b import other


@patch('module_a.func') ✓ Yes!
def test_other(mock_func):
    result = other()
    assert ...
```

Stubs

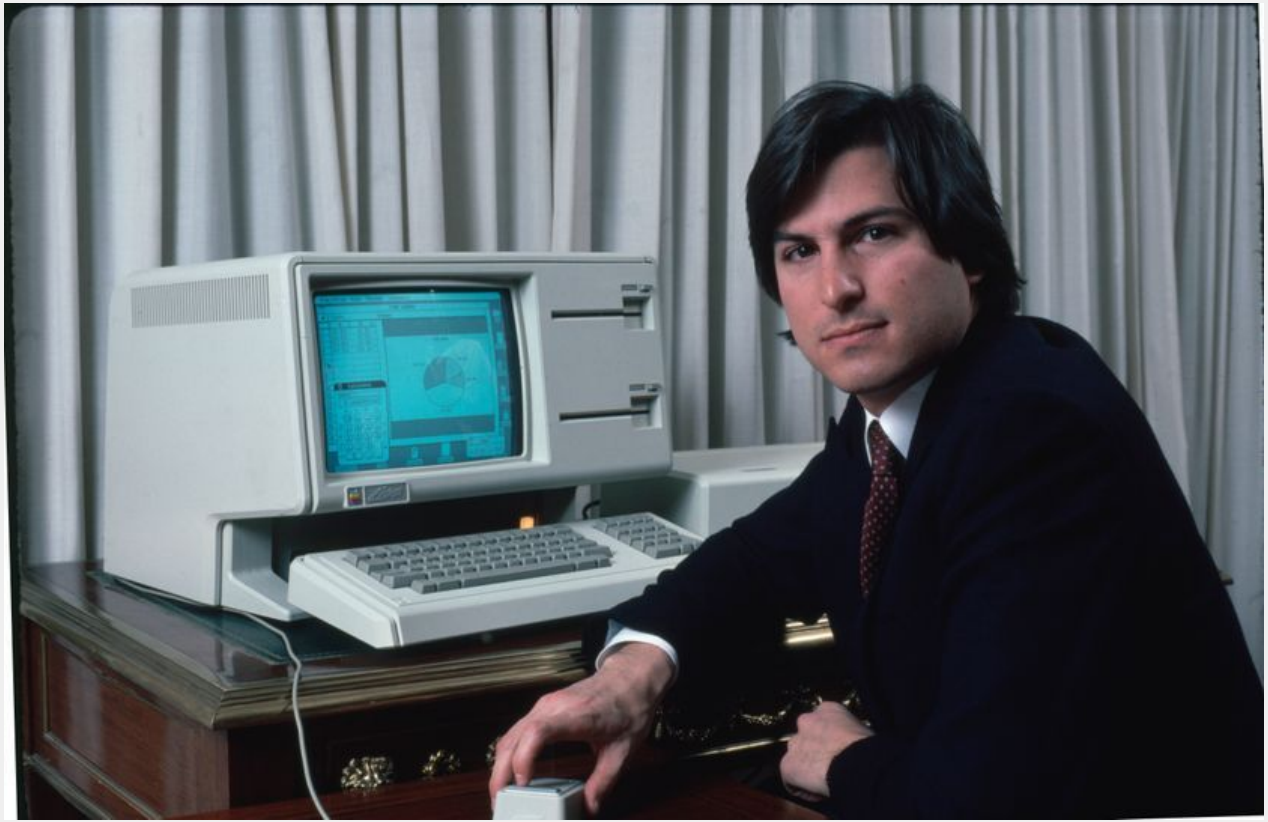
- Similar to Mocks...
- ...but already configured somewhere
- Mocks remove the need to create stubs everywhere

The `unittest.mock` module is **extremely rich!**

No chance to win without **reading the documentation** 😉

https://docs.python.org/3/library/unittest.mock.html

Oh, one more thing...



The AAA pattern

Arrange / Act / Assert

```
def test_my_add_function():
    a = 1
    b = 2

    result = my_add_function(a, b)

    assert result == a + b
```

Arrange

```
def test_my_add_function():
    a = 1
    b = 2

    result = my_add_function(a, b)

    assert result == a + b
```

Act

```
def test_my_add_function():
    a = 1
    b = 2

    result = my_add_function(a, b)

    assert result == a + b
```

Assert

```python
def test_my_add_function():
    a = 1
    b = 2

    result = my_add_function(a, b)

    assert result == a + b
```

And now… It’s your turn!