TESTING MULTI-OS PYTHON SOFTWARE

# TESTING MULTI-OS PYTHON SOFTWARE
INTRODUCTION

- Introduction
- Unit tests in python
- Automating tests
- General tips and techniques
ABOUT ME

• Developer job at open source software company uib GmbH, see https://uib.de
• Part of backend team at opsi Project, see https://opsi.org
• I love to automate things and make workflows simple, reproducible and robust
WHY SOFTWARE TESTING

- To find bugs (obviously)
- To not break things when refactoring code
- To be robust against malicious or stupid user input
WHY SOFTWARE TESTING

- To find bugs (obviously)
- To not break things when refactoring code
- To be robust against malicious or stupid user input

"Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning."

- Rick Cook
TYPES OF TESTING

- Unit testing: testing single components and core functionalities
- Integration testing: testing interfaces and collaboration of different components
- End-to-end testing: testing complete workflow over all involved components
TYPES OF TESTING

- Unit testing: testing single components and core functionalities
- Integration testing: testing interfaces and collaboration of different components
- End-to-end testing: testing complete workflow over all involved components

This talk focuses on unit testing python applications through python frameworks. Most concepts may be generalized.
UNIT TESTS IN PYTHON
PYTEST CONCEPT

- Command line tool to collect test scenarios from directory - see https://docs.pytest.org
- Execute tests one by one and evaluate results
- Human-readable summary in output (info of failed tests, percentage of test coverage per file)
- Generates line-by-line status file for tools
PYTEST IS USEFUL...

- For conveniently testing functionality of a whole project or part of it
- To parameterize tests (multiple inputs for the same function)
- When working with different environments (markers control which tests run where and how)
Given a function you want to test

```python
ROT13 = str.maketrans("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz", "NOPQRSTUVWXYZABCDEFGHIJKLMnopqrstuvwxyzabcdefghijklm")

def obfuscate(data: str) -> str:
    return data.translate(ROT13)
```

(It obfuscates a string by shifting each letter by 13 positions)
from testproject import obfuscate

def test_example() -> None:
    assert obfuscate("Hello World") == "Uryyb Jbeyq"
    assert obfuscate(obfuscate("teststring")) == "teststring"
from testproject import obfuscate

def test_example() -> None:
    assert obfuscate("Hello World") == "Uryyb Jbeyq"
    assert obfuscate(obfuscate("teststring")) == "teststring"
from testproject import obfuscate

import pytest

@pytest.mark.parametrize("data, result", (
    ("Hello World", "Uryyb Jbeyq"),
    ("The quick brown fox jumps over the lazy dog", "Gur dhvpx oebja sbk whzcf bire gur ynml qbt"),
    ("See you later!", "Frr lbh yngre!"),
))
def test_example2(data, result) -> None:
    assert obfuscate(data) == result
from testproject import obfuscate

import pytest

@pytest.mark.parametrize("data, result", ("Hello World", "Uryyb Jbeyq"),
("The quick brown fox jumps over the lazy dog", "Gur dhvpx oebja sbk whzcf bire gur ynml qbt"),
("See you later!", "Frr lbh yngre!"),
))
def test_example2(data, result) -> None:
    assert obfuscate(data) == result

# TESTING MULTI-OS PYTHON SOFTWARE
from testproject import obfuscate

import pytest

@pytest.mark.parametrize("data", (
    "Let's meet at the park."
    "My number is 0176123456789"
))
def test_example3(data) -> None:
    obfuscated = obfuscate(data)
    for word in data.split(" "):
        assert word not in obfuscated
# TESTING MULTI-OS PYTHON SOFTWARE

```python
platform linux -- Python 3.11.4, pytest-7.4.0, pluggy-1.2.0
rootdir: /home/doerr/Documents/ulb/testproject
plugins: hypothesis=6.81.1
collected 2 items

tests/test_pytest.py .F [100%]

================================================================== test_example3[My number is 0176123456789] ===================================================================

data = 'My number is 0176123456789'

@pytest.mark.parametrize("data", ("Let's meet at the park.", "My number is 0176123456789"))
def test_example3(data) -> None:
    obfuscated = obfuscate(data)
    for word in data.split(' '):
        assert word not in obfuscated

E   '0176123456789' not in 'Zl ahzore vF 0176123456789'
E   '0176123456789' is contained here:
E   Zl ahzore vF 0176123456789

tests/test_pytest.py:24:AssertionError

================================================================== short test summary info ===========

FAILED tests/test_pytest.py::test_example3[My number is 0176123456789] - AssertionError: assert '0176123456789' not in 'Zl ahzore vF 0176123456789'

================================================================== 1 failed, 1 passed in 0.02s ===========
```
A software tester walks into a bar. 
Crawls into a bar. 
Dances into a bar. 
Flies into a bar. 
And orders: 
a beer. 
99999999 beers. 
0 beers. 
a lizard in a beer glass. 
-1 beer. 
"qwertyuiop" beers.
HYPOTHESIS CONCEPT

A software tester walks into a bar.  
Crawls into a bar.  
Dances into a bar.  
Flies into a bar.  
And orders:  
a beer.  
99999999 beers.  
0 beers.  
a lizard in a beer glass.  
-1 beer.  
"qwertyuiop" beers.

A real customer walks into the bar and asks where the bathroom is.  
The bar goes up in flames.
HYPOTHESIS CONCEPT

- Define what types of input data are allowed for a function
- Check against arbitrary patterns following the scheme

Official documentation at https://hypothesis.readthedocs.io
HYPOTHESIS IS USEFUL...

- For identifying problematic input values
- For checking against extreme instances of some types
  - Integer-like: zero, negative numbers, extremely small and big numbers, ...
  - String-like: empty strings, long strings, special characters, ...
  - Path-like: relative and absolute paths, spaces, backslash and other atrocities in file names...
  - ...

# TESTING MULTI-OS PYTHON SOFTWARE
from hypothesis import given, strategies

from testproject import obfuscate

@given(strategies.characters())
def test_obfuscate_hyp(chars):
    assert obfuscate(obfuscate(chars)) == chars
    assert obfuscate(chars) != chars
platform linux -- Python 3.11.4, pytest-7.4.0, plugy-1.2.0
rootdir: /home/doerr/Documents/ub/testproject
plugins: hypothesis-6.81.1
collected 1 item

tests/test_hypothesis.py F [100%]

================================ FAILURES =================================

================================ test_obfuscate_hyp ========================

@given(strategies.characters())
>   def test_obfuscate_hyp(chars):

```
tests/test_hypothesis.py:14:
chars = '0'

@given(strategies.characters())
def test_obfuscate_hyp(chars):
    assert obfuscate(obfuscate(chars)) == chars
    assert obfuscate(chars) != chars
E   AssertionError: assert '0' != '0'
E   + where '0' = obfuscate('0')
E   Falsifying example: test_obfuscate_hyp(  
E     chars='0',
E   )
```

tests/test_hypothesis.py:16: AssertionError

--------------------------------- short test summary info ---------------------
FAILED tests/test_hypothesis.py::test_obfuscate_hyp - AssertionError: assert '0' != '0'

--------------------------------- 1 failed in 0.15s --------------------------
from hypothesis import given, strategies

given(strategies.integers(), strategies.integers())
def test ints are commutative(x, y):
    assert x + y == y + x
from hypothesis import given, strategies

@given(strategies.integers(), strategies.integers())
def test_ints_are_commutative(x, y):
    assert x + y == y + x

# TESTING MULTI-OS PYTHON SOFTWARE
from hypothesis import given, strategies

given(strategies.floats(), strategies.floats())
def test_floats_are_commutative(x, y):
    assert x + y == y + x
HYPOTHESIS EXAMPLE TEST 3

--- test session starts ---

platform linux -- Python 3.11.4, pytest-7.4.0, pluggy-1.2.0
rootdir: /home/doerrer/Documents/ib/testproject
plugins: hypothesis-6.81.1
collected 1 item

tests/test_hypothesis.py · F [100%]

--- failures ---

--- test_floats_are_commutative ---

@given(stategies.floats(), strategies.floats())
def test_floats_are_commutative(x, y):

x = inf, y = -inf

@given(stategies.floats(), strategies.floats())
def test_floats_are_commutative(x, y):
    assert x + y == y + x
    assert (inf + -inf) == (-inf + inf)
E Falsifying example: test_floats_are_commutative(
    x=inf,
    y=-inf,
E )

1 failed in 0.21s

--- short test summary info ---

FAILED tests/test_hypothesis.py: test_floats_are_commutative · assert (inf + -inf) == (-inf + inf)
AUTOMATING TESTS
Git is a powerful tool!

Great open source software development platform gitlab - see https://about.gitlab.com

Feature gitlab-ci performs pipeline of defined jobs at each "git push"

Job consists of actions - executed consecutively as long as exit code is 0 (OK)

Talk of Erol Ülükmen at FrOSCon 2022: https://media.ccc.de/v/froscon2022-2795-ci_first
Pytest has exit code 1 if one test fails.

So gitlab-ci job running pytest automatically fails on a test fail.

Further steps are not triggered (shipping to internal end-to-end testing environment or even production)

Author of commit is notified
MULTI-PLATFORM PYTHON APPS

- Easy to build multi-platform apps with python
- Development on one platform, then shipped to many
- Fundamental differences between platforms
  - Config files vs. registry
  - Handling of network shares and other resources
  - Users, file permissions, ...
  - ...

# TESTING MULTI-OS PYTHON SOFTWARE
Testing on Multiple Platforms

- Some functionality can be mocked - e.g. simulating a web server
- Certainty only if testing behavior on (all!) target platforms
- Linux-like systems available in docker
- Windows, MacOS systems as VMs
COLLECTING TEST RESULTS

- One gitlab-ci job per platform (running in container/VM)
- Each pytest creates coverage file (xml-format) and stores it as pipeline artifact
- Final job collects artifacts and combines coverage files (by using "coverage" tool - see https://coverage.readthedocs.io)
- Result is human-readable summary and machine-readable coverage file

# TESTING MULTI-OS PYTHON SOFTWARE
# TESTING MULTI-OS PYTHON SOFTWARE

## COLLECTING TEST RESULTS

Example summary for "opsi-cli"

<table>
<thead>
<tr>
<th>Name</th>
<th>Stmts</th>
<th>Miss</th>
<th>Cover</th>
<th>Miss</th>
<th>Cover (Windows only)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>opsicli/__init__.py</code></td>
<td>11</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td><code>opsicli/__main__.py</code></td>
<td>105</td>
<td>9</td>
<td>91%</td>
<td>9</td>
<td>91%</td>
</tr>
<tr>
<td><code>opsicli/cache.py</code></td>
<td>61</td>
<td>4</td>
<td>93%</td>
<td>7</td>
<td>89%</td>
</tr>
<tr>
<td><code>opsicli/config.py</code></td>
<td>310</td>
<td>35</td>
<td>89%</td>
<td>41</td>
<td>87%</td>
</tr>
<tr>
<td><code>opsicli/io.py</code></td>
<td>272</td>
<td>44</td>
<td>84%</td>
<td>45</td>
<td>83%</td>
</tr>
<tr>
<td><code>opsicli/messagebus.py</code></td>
<td>136</td>
<td>35</td>
<td>74%</td>
<td>106</td>
<td>22%</td>
</tr>
<tr>
<td><code>opsicli/opsiservice.py</code></td>
<td>75</td>
<td>29</td>
<td>61%</td>
<td>58</td>
<td>23%</td>
</tr>
<tr>
<td><code>opsicli/plugin.py</code></td>
<td>181</td>
<td>10</td>
<td>94%</td>
<td>10</td>
<td>94%</td>
</tr>
<tr>
<td><code>opsicli/types.py</code></td>
<td>110</td>
<td>10</td>
<td>91%</td>
<td>12</td>
<td>89%</td>
</tr>
<tr>
<td><code>opsicli/utils.py</code></td>
<td>62</td>
<td>21</td>
<td>66%</td>
<td>23</td>
<td>63%</td>
</tr>
<tr>
<td><code>plugins/client-action/python/__init__.py</code></td>
<td>37</td>
<td>0</td>
<td>100%</td>
<td>4</td>
<td>89%</td>
</tr>
<tr>
<td><code>plugins/client-action/python/client_action_worker.py</code></td>
<td>32</td>
<td>3</td>
<td>91%</td>
<td>25</td>
<td>22%</td>
</tr>
<tr>
<td><code>plugins/client-action/python/set_action_request_worker.py</code></td>
<td>125</td>
<td>19</td>
<td>85%</td>
<td>111</td>
<td>11%</td>
</tr>
<tr>
<td><code>plugins/config/python/__init__.py</code></td>
<td>115</td>
<td>27</td>
<td>77%</td>
<td>27</td>
<td>77%</td>
</tr>
<tr>
<td><code>plugins/jsonrpc/python/__init__.py</code></td>
<td>76</td>
<td>22</td>
<td>71%</td>
<td>47</td>
<td>38%</td>
</tr>
<tr>
<td><code>plugins/manager-repo/python/__init__.py</code></td>
<td>106</td>
<td>4</td>
<td>96%</td>
<td>4</td>
<td>96%</td>
</tr>
<tr>
<td><code>plugins/plugin/python/__init__.py</code></td>
<td>164</td>
<td>27</td>
<td>84%</td>
<td>27</td>
<td>84%</td>
</tr>
<tr>
<td><code>plugins/self/plugin/__init__.py</code></td>
<td>152</td>
<td>36</td>
<td>76%</td>
<td>63</td>
<td>59%</td>
</tr>
<tr>
<td><code>plugins/support/python/__init__.py</code></td>
<td>42</td>
<td>12</td>
<td>71%</td>
<td>16</td>
<td>62%</td>
</tr>
<tr>
<td><code>plugins/support/python/worker.py</code></td>
<td>10</td>
<td>0</td>
<td>100%</td>
<td>7</td>
<td>30%</td>
</tr>
<tr>
<td><code>plugins/terminal/python/__init__.py</code></td>
<td>22</td>
<td>3</td>
<td>86%</td>
<td>3</td>
<td>86%</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td>2260</td>
<td>340</td>
<td>85%</td>
<td>646</td>
<td>71%</td>
</tr>
</tbody>
</table>
GENERAL TIPS AND TECHNIQUES
TEST DRIVEN DEVELOPMENT

Reverse classical testing workflow:

1. Think about tasks and how to wrap them in classes and functions
2. Write tests for each class and function
3. Implement the actual classes and functions so that tests do not fail
USE GIT BRANCHES

- Main branch always ready for release
- Individual feature- and fix- branches, merge request to main once complete and tested
- Merge request reviewed and accepted/denied by another developer (principle of 4 eyes)
USE VIRTUAL ENVIRONMENTS

- Virtual python environments are isolated from host system
- Handle dependency management tailored to specific application
- If you break it - you don't break your host system
- Some frameworks assisting with that:
  - venv: very basic
  - conda environments: convenient
  - poetry: comfortable and flexible
USEFUL IDE FEATURES

- Dev-container: isolated environment (tests do not affect host)
- Go-to-definition (e.g. CTRL + click): shows origin of selected component
- Coverage gutter: pytest coverage line by line
- Git marker: added, changed, deleted lines visualized

```python
class File(type(Path())):
    
    def __new__(cls, *args, **kwargs):
        file = super().__new__(cls, *args, **kwargs)
        path = self._path.expanduser().absolute()
        if not path.exists() and not path.is_file():
            raise ValueError(f"Not a file: {path!r}"
        return path
```
USE LINTERS

- Linters highlight potential problems of all sorts
- Some may even automatically fix small issues
  - pylint and ruff: check general python conventions
  - flake8: checks code style
  - mypy: checks data types
Python is dynamically typed - variables can point to objects of different types.
Type hints help to keep types in mind

```python
def get_length(value):
    return len(value)

print(get_length("teststring"))  # ok
print(get_length("1234"))        # ok
print(get_length(1234))          # exception at runtime
```

# TESTING MULTI-OS PYTHON SOFTWARE
USE TYPE HINTS

- Python is dynamically typed - variables can point to objects of different types.
- Type hints help to keep types in mind

```python
def get_length(value):
    return len(value)

print(get_length("teststring"))  # ok
print(get_length("1234"))       # ok
print(get_length(1234))         # exception at runtime
```

```python
def get_length(value: str) -> int:
    return len(value)

print(get_length("teststring"))  # ok
print(get_length("1234"))        # ok
print(get_length(1234))          # IDE will probably warn you
```
DO NOT REINVENT THE WHEEL

- Use recent python release for newest python standard library features
- Python has a large community with thousands of cool libraries
- Catalogue at https://pypi.org
- If you spot a problem: open an issue! Don’t just workaround
THANK YOU FOR YOUR ATTENTION!

Questions?
ADDITIONAL MATERIAL: SETTING UP A NEW PROJECT WITH POETRY

- poetry is a python dependency manager - see https://python-poetry.org/
- poetry new <name> creates
  - pyproject.toml with general project info and dependency list
  - <name> directory with empty __init__.py file (python package)
  - tests directory with empty `__init__.py` file (package for the tests)
- poetry install creates virtual environment
ADDITIONAL MATERIAL: JENKINS

- Open source automation system [https://jenkins.io](https://jenkins.io)
- We use it for end-to-end tests:
  - Triggered by gitlab-ci or manual action
  - Jobs run on VMs, packages collected from development repository
  - Script controls how components interact