{Never write scripts again}

Python CLI applications & TDD

Swiss Python Summit 2022

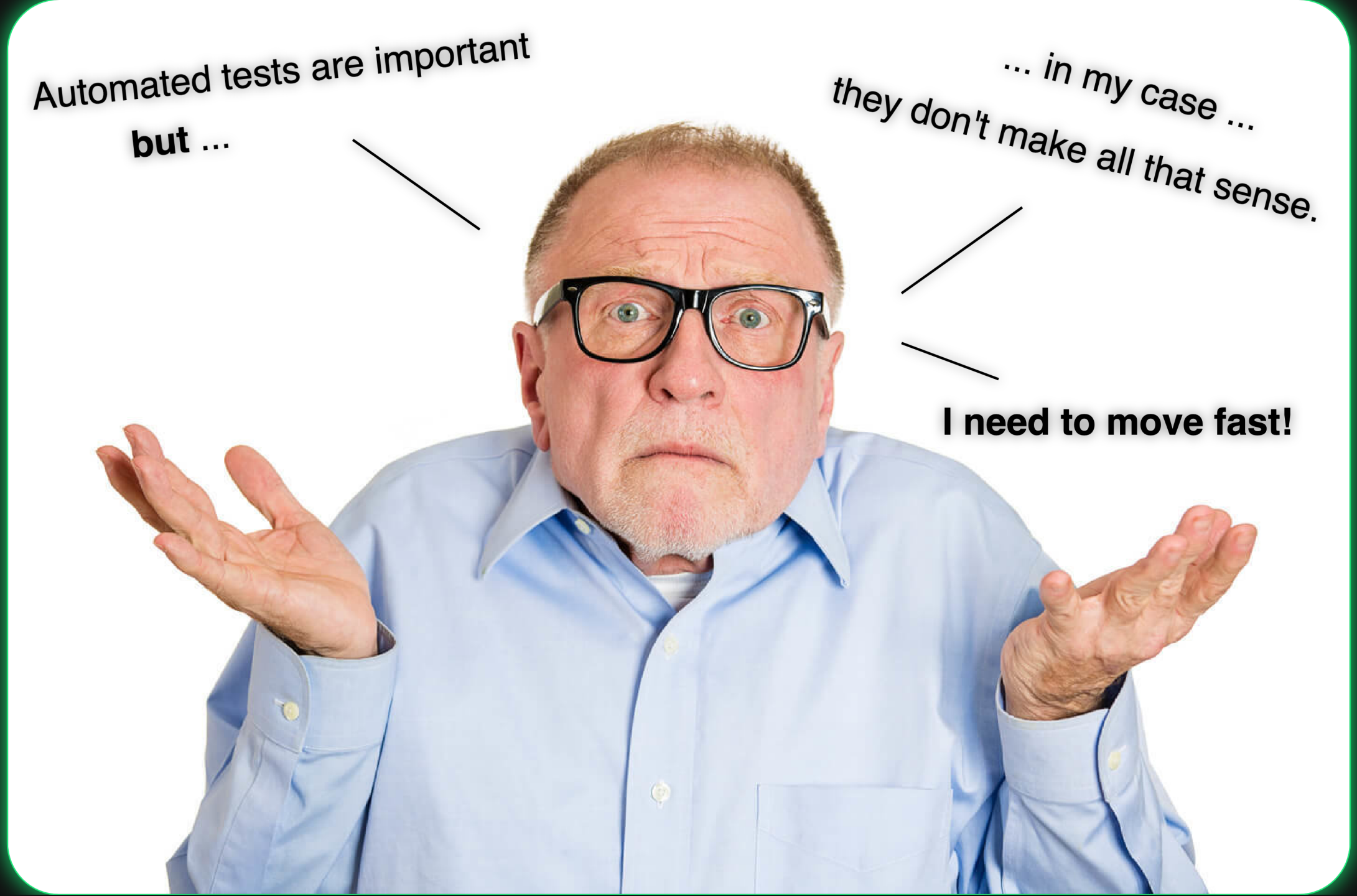
{ Peter Bittner }

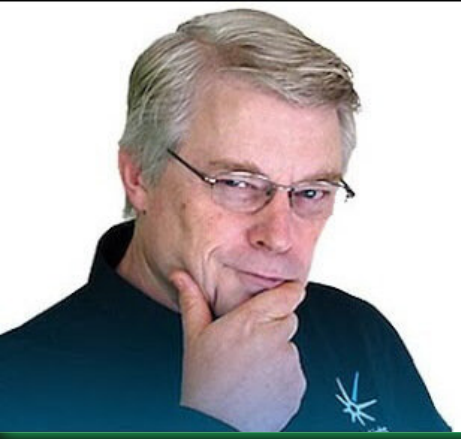
Developer
of people, companies and code

@peterbittner, django@bittner.it

swisscom

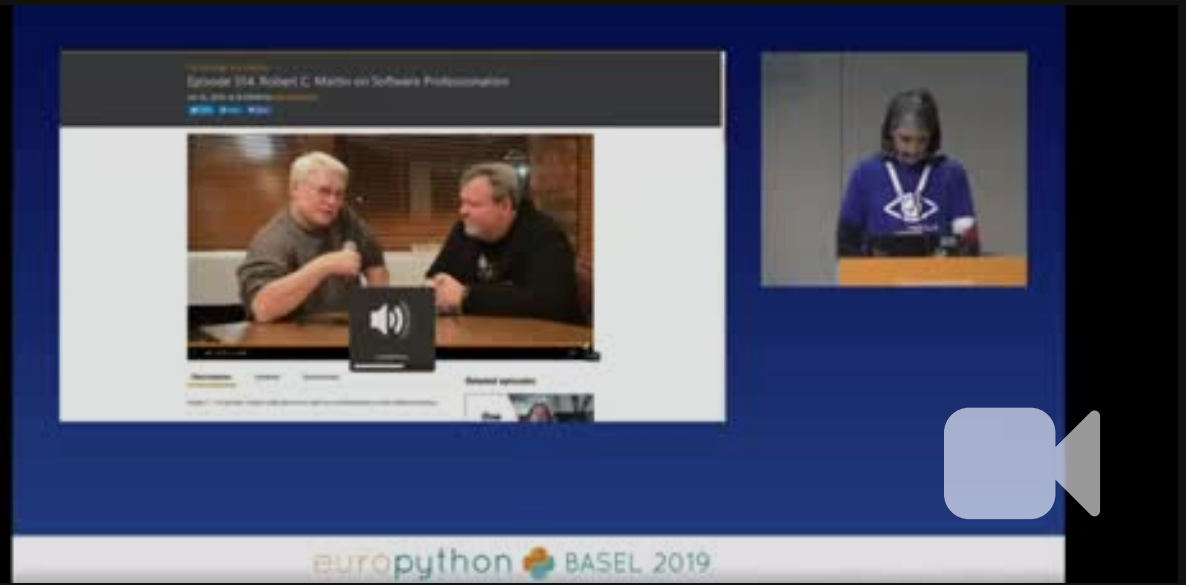
cli-test-helpers PythonTurtle ansible-role-software
behave-django django-analytical djangocms-maps django-probes
django-bootstrap-static django-apptemplates django-organice pyclean


Automated tests are important
but ...
... in my case ...
they don't make all that sense.
I need to move fast!



“ The only way to go fast
is to go well.

--- Robert C. Martin



Source: Technology and Friends, Episode 354, 2015

AGENDA

CLI Applications & TDD

1. What's wrong with scripts?
2. *Coding example* #1 (refactoring)
3. Why CLI applications?
4. Challenges with writing tests
5. *Coding example #2* (cli & tests)

```python
from application import cli
from cli_test_helpers import shell

def test_cli_entrypoint():
    result = shell("python-summit --help")
    assert result.exit_code == 0
```

What's Wrong with Scripts?

```python
print("This is important code")

for index, arg in enumerate(sys.argv):
    print(f"[{index}]: {arg}")
```

- Easy to get started
- Limited possibilities for structure
- Hard to (unit) test
- No dependency management
- Deployment may require care
- Custom user experience

Coding Example #1

```
$ cowsay -e ~~ WHAT IS WRONG WITH SCRIPTS? | lolcat
 _____________________________
< WHAT IS WRONG WITH SCRIPTS? >
 -----------------------------
        \   ^__^
         \  (~~)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Refactor a script

Why CLI Applications?

```python
def main():
    print("Usage: foo <bar> --baz")

if __name__ == "__main__":
    main()
```

- Standardized user experience
- More possibilities for structure
- Possibilities for all kinds of testing
- Dependency management
- Packaging & distribution

Argparse

```
import argparse
from . import __version__

def parse_arguments():
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument('--version', action='version',
                        version=__version__)
    parser.add_argument('filename')
    args = parser.parse_args()
    return args

def main():
    args = parse_arguments()
    ...
```

Click

```
1 import click
2
3 @click.command()
4 @click.version_option()
5 @click.argument('infile', type=click.File())
6 def main(infile):
7     click.echo(infile.read())
```

Docopt

```
"""Foobar
Usage:
  foobar (-h | --help | --version)
  foobar [-s | --silent] <file>
  foobar [-v | --verbose] <file>
Positional arguments:
  file           target file path name
Optional arguments:
  -h, --help     show this help message and exit
  -s, --silent   don't show progress output
  -v, --verbose  explain progress verbosely
  --version      show program's version number and exit
"""
from docopt import docopt
from . import __version__

def parse_arguments():
    args = docopt(__doc__, version=__version__)

    return dict(
        file=args['<file>'],
        silent=args['-s'] or args['--silent'],
        verbose=args['-v'] or args['--verbose'],
    )
```

Challenges with Writing Tests

```python
def test_cli():
    with pytest.raises(SystemExit):
        foobar.cli.main()
        pytest.fail("CLI doesn't abort")
```

- How test our CLI configuration?
- Control taken away from us
- Package features require deployment

CLI Test Strategy

```python
def a_functional_test():
    result = shell('foobar')
    assert result.exit_code != 0, result.stdout

def a_unit_test():
    with ArgvContext('foobar', 'myfile', '--verbose'):
        args = foobar.cli.parse_arguments()
    assert args['verbose'] == True
```

1. Start from the user interface with **functional tests**.
2. Work down **towards unit tests**.

Functional Tests (User Interface)

```python
def test_entrypoint():
    """Is entrypoint script installed? (setup.py)"""
    result = shell('foobar --help')
    assert result.exit_code == 0
```

Drive the code from "outside"

"Is the **entrypoint script** installed?"

"Can package be **run as a** Python **module**?"

"Is **positional** argument <*foo*> available?"

"Is **optional** argument *--bar* available?"

Unit Tests

```python
import foobar
from cli_test_helpers import ArgvContext
from unittest.mock import patch

@patch('foobar.command.process')
def test_process_is_called(mock_command):
    with ArgvContext('foobar', 'myfile', '-v'):
        foobar.cli.main()

    assert mock_command.called
    assert mock_command.call_args.kwargs == dict(
        file='myfile', silent=False, verbose=True)
```

Stay in the Python code

Tox

Installs your CLI before running the test suite!

```
[tox]
envlist = py{38,39,310}

[testenv]
description = Unit tests
deps =
    cli-test-helpers
    coverage[toml]
    pytest
commands =
    coverage run -m pytest {posargs}
    coverage xml
    coverage report
```

Coding Example #2

```
$ cowsay MOO! I ♥ CLI-TEST-HELPERS | lolcat
 ____________________________
< MOO! I ♥ CLI-TEST-HELPERS >
 ----------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Write and run tests with `cli-test-helpers`

Thank you!

for your precious time

Painless Software

Less pain, more fun.