Go through the README.md of:

github.com/Vantage-AI/testing_for_ds_day_0_vantage

/

github.com/BigDataRepublic/testing_for_ds_day_0_bdr

Testing for Data Science

Marks Fixtures Parametrize Error handling Temporary paths

The example

```python
import pytest
import time


@pytest.mark.slow
def test_patience():
    time.sleep(10)
    assert "patience" not in "programmer"
```

```
$ pytest -m "slow" tests/test_marks.py
========================== test session starts ==========================
platform win32 -- Python 3.7.11, pytest-6.2.4, py-1.11.0, pluggy-0.13.1
cachedir: tests\.pytest_cache
rootdir: C:\Users\...\testing_for_data_science, configfile:
pyproject.toml
plugins: cov-2.9.0, mock-3.6.1
collected 1 item

tests\test_marks.py .                                                  [100%]

=========================== warnings summary ============================
tests\test_marks.py:5

C:\Users\jaspe\Documents\bdr\kennisbank\testing_for_data_science\tests
\test_marks.py:5: PytestUnknownMarkWarning: Unknown pytest.mark.slow -
is this a typo?  You can register custom marks to avoid this warning -
for details, see https://docs.pytest.org/en/stable/mark.html
    @pytest.mark.slow

-- Docs: https://docs.pytest.org/en/stable/warnings.html
==================== 1 passed, 1 warning in 10.04s ====================
```

Register marks

```toml
# Contents of pyproject.toml.

[tool.pytest.ini_options]
markers = [
        "slow: mark slow tests (deselect with '-m \"not slow\"') ",
]
```

```
$ pytest -m "not slow" tests/test_marks.py
========================== test session starts ==========================
platform win32 -- Python 3.7.11, pytest-6.2.4, py-1.11.0, pluggy-0.13.1
cachedir: tests\.pytest_cache
rootdir: C:\...\testing_for_data_science, configfile: pyproject.toml
plugins: cov-2.9.0, mock-3.6.1
collected 1 item / 1 deselected

======================== 1 deselected in 0.02s ========================
```

skip
For whatever reason or based on some condition.

The example

```
@pytest.mark.skip(reason="We are not patient.")
def test_patience_again():
    time.sleep(10)
    assert "patience" not in "programmer"
```

```
$ pytest -m "not slow" -rs tests/test_marks.py
========================= test session starts =========================
platform win32 -- Python 3.7.11, pytest-6.2.4, py-1.11.0, pluggy-0.13.1
cachedir: tests\.pytest_cache
rootdir: C:\...\testing_for_data_science, configfile: pyproject.toml
plugins: cov-2.9.0, mock-3.6.1
collected 2 items / 1 deselected / 1 selected

tests\test_marks.py s                                                [100%]

====================== short test summary info ======================
SKIPPED [1] tests\test_marks.py:11: We are not patient.
================== 1 skipped, 1 deselected in 0.01s ==================
```

Skip conditionally

```
@pytest.mark.skipif(condition=sys.version_info > (2, 7), reason="Tests
for older python version.")
def test_for_python2_7():
    assert sys.version_info <= (2, 7)
```

Skip all tests in a module

```
# test_module.py
pytestmark = pytest.mark.skipif(condition="..." > 0, reason="...")
```

Xfail

Mark tests that are expected to fail, for instance if a feature isn't implemented yet. If it passes nevertheless it's called an xpass but this is ignored by default.

example

```python
import pytest


@pytest.mark.xfail
def test_predict_fortune():
    future = predict_future(person="Jasper")  # not implemented
    assert future == "good"
```

```
$ pytest tests/test_marks_xfail.py
========================= test session starts ==========================
platform win32 -- Python 3.7.11, pytest-6.2.4, py-1.11.0, pluggy-0.13.1
cachedir: tests\.pytest_cache
rootdir: C:\...\testing_for_data_science, configfile: pyproject.toml
plugins: cov-2.9.0, mock-3.6.1
collected 1 item

tests\test_marks_xfail.py x                                      [100%]

========================== 1 xfailed in 0.04s ==========================
```

The example

```python
@pytest.mark.xfail(reason="Not implemented",
                   raises=NameError,
                   run=False,
                   strict=True)
def test_predict_fortune():
    future = predict_future(person="Jasper")
    assert future == "good"
```

Marks summary

```
# Filter tests on marks with ```pytest -m "slow"```
# Display all registerd markers with ```pytest --markers```
@pytest.mark.slow


# Markers with additional functionality
@pytest.mark.skip(reason="...")
@pytest.mark.skipif(condition=..., reason="...")
@pytest.mark.xfail(reason="Not implemented",
                   raises=NameError, run=False, strict=True)


# On module level
pytestmark = pytest.mark.skipif(condition="..." > 0, reason="...")
```

```python
# contents of test_employee.py
from src.data_model import Employee
import pytest

def test_employee_name():
    employee_anna = Employee(name="Anna", age=37)
    assert employee_anna.name == "Anna"


def test_employee_give_birthday():
    employee_anna = Employee(name="Anna", age=37)
    employee_anna.give_birthday()
    assert employee_anna.age == age + 1
```

```python
# contents of test_employee.py
from src.data_model import Employee
import pytest


@pytest.fixture
def employee_anna():
    return Employee(name="Anna", age=37)


def test_employee_name(employee_anna):
    assert employee_anna.name == "Anna"


def test_employee_give_birthday(employee_anna):
    initial_age = employee_anna.age
    employee_anna.give_birthday()
    assert employee_anna.age == initial_age + 1
```

Fixtures requesting (same) fixtures

```python
# contents of test_append.py
import pytest

@pytest.fixture
def first_entry():
    return "a"


@pytest.fixture
def order():
    return []


@pytest.fixture
def append_first(order, first_entry):
    return order.append(first_entry)


def test_append(append_first, order, first_entry):
    assert append_first == [first_entry]
    assert order == []
    assert order == ["a"]
```

Dependency tree

```python
# contents of test_setup_teardown.py
import pytest


@pytest.fixture(scope="function")
def fixture_0():
    print("SETUP fixture_0")
    yield "string_0"
    print("TEARDOWN fixture_0")


@pytest.fixture(scope="module")
def fixture_1():
    print("SETUP fixture_1")
    yield "string_1"
    print("TEARDOWN fixture_1")


def test_0(fixture_0):
    print(f"RUN test0, f_0: {fixture_0}")


def test_1(fixture_1):
    print(f"RUN test1, f_1: {fixture_1}")


def test_2(fixture_0, fixture_1):
    print(f"RUN test2, f_0: {fixture_0}, f_1: {fixture_1}")
```

```
$ pytest -s tests/test_setup_teardown.py
========================== test session starts ==========================
platform win32 -- Python 3.7.11, pytest-6.2.4, py-1.11.0, pluggy-0.13.1
cachedir: tests\.pytest_cache
rootdir: C:\...\testing_for_data_science, configfile: pyproject.toml
plugins: cov-2.9.0, mock-3.6.1
collected 3 items

tests\test_fixtures.py
SETUP fixture_0
  RUN test0 with fixture_0: string_0
TEARDOWN fixture_0
SETUP fixture_1
  RUN test1 with fixture_1: string_1
SETUP fixture_0
  RUN test2 with fixture_0: string_0 and fixture_1: string_1
TEARDOWN fixture_0
TEARDOWN fixture_1

========================== 3 passed in 0.04s ============================
```

Possible but not desirable

```python
@pytest.fixture
def receiving_user(mail_admin, request):
    user = mail_admin.create_user()

    def delete_user():
        mail_admin.delete_user(user)

    request.addfinalizer(delete_user)
    return user
```

Use fixtures with mark

For instance all methods of a class require a fixture.
This doesn't work for requiring fixtures in fixtures.

class

```python
# content of test_use_fixture_with_mark.py
import os
import tempfile
import pytest


@pytest.fixture
def clean_dir():
    with tempfile.TemporaryDirectory() as newpath:
        old_cwd = os.getcwd()
        os.chdir(newpath)
        yield
        os.chdir(old_cwd)


@pytest.mark.usefixtures("clean_dir")
class TestDirectoryInit:
    def test_cwd_starts_empty(self):
        with open("myfile", "w") as f:
            f.write("hello")
        assert os.listdir(os.getcwd()) == ["myfile"]

    def test_cwd_starts_empty_again(self):
        assert os.listdir(os.getcwd()) == []
```

module

```python
pytestmark = pytest.mark.usefixtures("clean_dir")
```

Possible values are **"function"** *(default),* **"class"**, **"module"** *and* **"package"**.

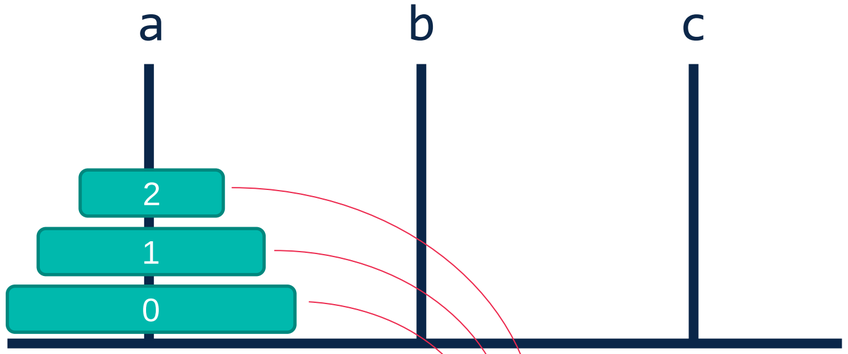
Good practice to explicitly request fixtures and avoid autouse.

The example

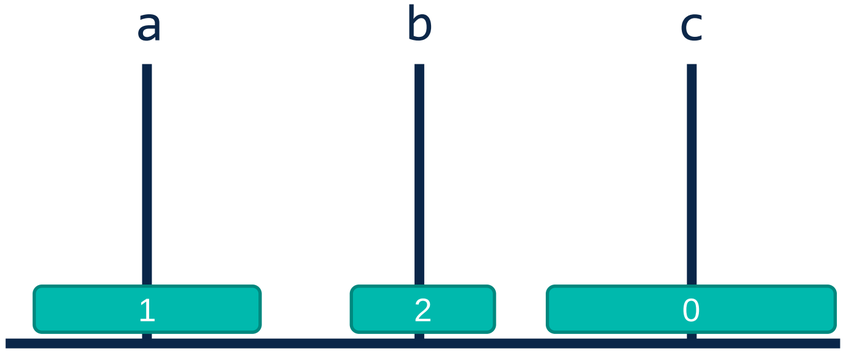
```python
@pytest.fixture(scope="session", autouse=True)
def summary_fixture():
    """Add a docstring because it shows up when running ```pytest --fixtures```."""
    # Put fixture setup here.
    yield "string_0"
    # Put fixture teardown here.
```

Use **yield** *instead of* **return**.

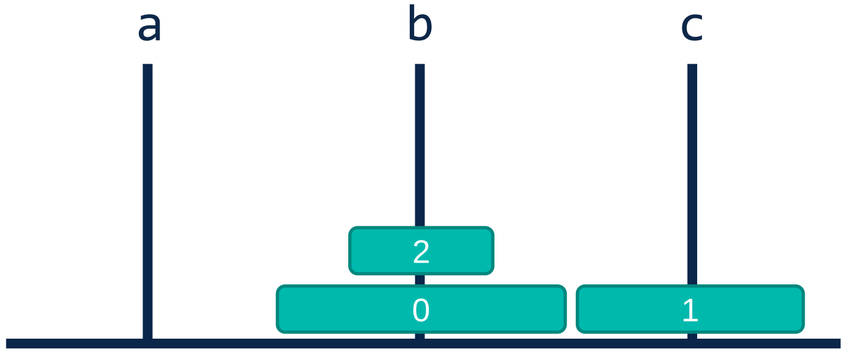
Keep fixtures small, to ensure save teardown.



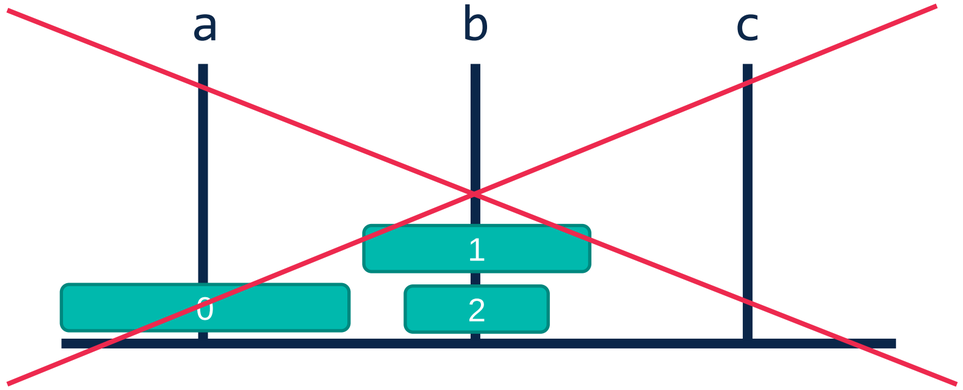
Position: "aaa"



Position: "cab"



Position: "bcb"



With a mark for a test

```
@pytest.mark.parametrize("n, expected", [(1, 2), (3, 4)])
class TestClass:
    def test_simple_case(self, n, expected):
        assert n + 1 == expected

    def test_weird_simple_case(self, n, expected):
        assert (n * 1) + 1 == expected
```

For a fixture

```
@pytest.fixture(scope="module", params=["smtp.gmail.com", "mail.python.org"])
def smtp_connection(request):
    smtp_connection = smtplib.SMTP(request.param, 587, timeout=5)
    yield smtp_connection
    smtp_connection.close()
```

Mark individual test instances

```
@pytest.mark.parametrize(
    ("n", "expected"),
    [(1, 1),
     (3, 6),
     pytest.param(40000, 0, marks=pytest.mark.xfail(raises=RecursionError))])
def test_factorial(n, expected):
    assert factorial(n) == expected


def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

Test for expected exceptions

factorial.py

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

test_factorial.py

```
import pytest


@pytest.mark.parametrize(
    ("n", "expected"),
    [(1, 1),
     (3, 6),
     pytest.param(40000, 0,
marks=pytest.mark.xfail(raises=RecursionError))])
def test_factorial(n, expected):
    assert factorial(n) == expected
```

```
import pytest


@pytest.mark.parametrize(("n", "expected"), [(1, 1), (3, 6)])
def test_factorial(n, expected):
    assert factorial(n) == expected


def test_factorial_recursionerror():
    with pytest.raises(expected_exception=RecursionError):
        factorial(40000)
```

Test for expected exceptions

factorial.py

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

test_factorial.py

```
import pytest


@pytest.mark.parametrize(("n", "expected"), [(1, 1), (3, 6)])
def test_factorial(n, expected):
    assert factorial(n) == expected


def test_factorial_recursionerror():
    with pytest.raises(expected_exception=RecursionError):
        factorial(40000)


@pytest.mark.parametrize("n", [0.4, "hallo", True])
def test_factorial_type(n):
    with pytest.raises(expected_exception=TypeError):
        factorial(n)
```

Test for expected exceptions

factorial.py

```
def factorial(n):
    if type(n) != int:
        raise TypeError("This is the wrong type!")
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

test_factorial.py

```
import pytest


@pytest.mark.parametrize(("n", "expected"), [(1, 1), (3, 6)])
def test_factorial(n, expected):
    assert factorial(n) == expected


def test_factorial_recursionerror():
    with pytest.raises(expected_exception=RecursionError):
        factorial(40000)


@pytest.mark.parametrize("n", [0.4, "hallo", True])
def test_factorial_type(n):
    with pytest.raises(expected_exception=TypeError):
        factorial(n)
```

Test for expected exceptions

factorial.py

```
def factorial(n):
    if type(n) != int:
        raise TypeError("This is the wrong type!")
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

test_factorial.py

```
import pytest


@pytest.mark.parametrize(("n", "expected"), [(1, 1), (3, 6)])
def test_factorial(n, expected):
    assert factorial(n) == expected


def test_factorial_recursionerror():
    with pytest.raises(expected_exception=RecursionError):
        factorial(40000)


@pytest.mark.parametrize("n", [0.4, "hallo", True])
def test_factorial_type(n):
    with pytest.raises(expected_exception=TypeError) as exception_info:
        factorial(n)
    assert "int" in str(exception_info.value)
```

Test for expected exceptions

factorial.py

```
def factorial(n):
    if type(n) != int:
        raise TypeError("Input of factorial should be of type 'int'.")
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

test_factorial.py

```
import pytest


@pytest.mark.parametrize(("n", "expected"), [(1, 1), (3, 6)])
def test_factorial(n, expected):
    assert factorial(n) == expected


def test_factorial_recursionerror():
    with pytest.raises(expected_exception=RecursionError):
        factorial(40000)


@pytest.mark.parametrize("n", [0.4, "hallo", True])
def test_factorial_type(n):
    with pytest.raises(expected_exception=TypeError) as exception_info:
        factorial(n)
    assert "int" in str(exception_info.value)
```

Test for expected exceptions

factorial.py

```python
def factorial(n):
    if type(n) != int:
        raise TypeError("Input of factorial should be of type 'int'.")
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

Also works but PyTest documentation advices to use context manager because it's more readable and less error prone.

test_factorial.py

```python
import pytest


@pytest.mark.parametrize(("n", "expected"), [(1, 1), (3, 6)])
def test_factorial(n, expected):
    assert factorial(n) == expected


def test_factorial_recursionerror():
    with pytest.raises(expected_exception=RecursionError):
        factorial(40000)


@pytest.mark.parametrize("n", [0.4, "hallo", True])
def test_factorial_type(n):
    with pytest.raises(expected_exception=TypeError) as exception_info:
        factorial(n)
    assert "int" in str(exception_info.value)


@pytest.mark.parametrize("n", range(-1, -20, -2))
def test_factorial_negative_int(n):
    pytest.raises(ValueError, factorial, **{"n": n})
```

test_temporary_paths.py

```python
def test_write_file(tmp_path):
    with open(os.path.join(tmp_path, "file.txt"), "w") as f:
        f.write("contents")
    assert len(list(tmp_path.iterdir())) == 1
```

pyproject.toml

```
[tool.pytest.ini_options]
markers = [
"slow: marks tests as slow (deselect with '-m \"not slow\"')",
]
addopts = "--basetemp tests/pytest_basetemp"
```

Folder structure

```
├── src
│   ├── data.py
│   └── model.py
├── tests
│   ├── pytest_basetemp
│   │   └── test_write_file0
│   │       └── file.txt
│   └── test_temporary_paths.py
└── pyproject.toml
```

tmp_path & tmp_path_factory

- Are fixtures of scope “function”.
- Generate temporary paths for tests.
- The last 3 versions of these paths are stored in your systems temporary directory.
- Unless you specify the location with ```addopts = "--basetemp tests/pytest_basetemp"``` (then only the last version is stored).
- tmp_dir & tmp_dir_factory are deprecated.

test_temporary_paths.py

```python
def test_write_file(tmp_path):
    with open(os.path.join(tmp_path, "file.txt"), "w") as f:
        f.write("contents")
    assert len(list(tmp_path.iterdir())) == 1


def test_write_files(tmp_path_factory):
    for char in ["a", "b", "c"]:
        path = tmp_path_factory.mktemp("factory_output")
        with open(os.path.join(path, f"{char*3}.txt"), "w") as f:
            f.write("contents")
        assert f"{char*3}.txt" in listdir(path)
```

pyproject.toml

```
[tool.pytest.ini_options]
markers = [
"slow: marks tests as slow (deselect with '-m \"not slow\"')",
]
addopts = "--basetemp tests/pytest_basetemp"
```

Folder structure

```
├── src
│   ├── data.py
│   └── model.py
├── tests
│   ├── pytest_basetemp
│   │   ├── factory_output0
│   │   │   └── aaa.txt
│   │   ├── factory_output1
│   │   │   └── bbb.txt
│   │   ├── factory_output2
│   │   │   └── ccc.txt
│   │   └── test_write_file0
│   │       └── file.txt
│   └── test_temporary_paths.py
└── pyproject.toml
```

Marks Fixtures Parametrize Error handling Temporary paths

fixtures

```python
@pytest.fixture(scope="session", autouse=True) # Try to avoid autouse.
def summary_fixture():
    """Add a docstring. Shows up with ```pytest --fixtures```."""
    # Put fixture setup here.
    yield "string_0"
    # Put fixture teardown here.
```

marks

```python
# Filter tests on marks with ```pytest -m "slow"```
# Display all registerd markers with ```pytest --markers```
@pytest.mark.slow


# Markers with additional functionality
@pytest.mark.skip(reason="...")
@pytest.mark.skipif(condition=..., reason="...")
@pytest.mark.xfail(reason="Not implemented",
                   raises=NameError, run=False, strict=True)
@pytest.mark.usefixtures("clean_dir")


# On module level
pytestmark = pytest.mark.skipif(condition="..." > 0, reason="...")
pytestmark = pytest.mark.usefixtures("clean_dir", "another_fixture")
```

parametrize

```python
@pytest.mark.parametrize("n, expected", [(1, 2), (3, 4)])
def test_simple_case(n, expected):
    assert n + 1 == expected

@pytest.fixture(params=["smtp.gmail.com", "mail.python.org"])
def smtp_connection(request):
    smtp_connection = smtplib.SMTP(request.param, 587, timeout=5)
    yield smtp_connection
    smtp_connection.close()
```

error handling

```python
def test_factorial_recursionerror():
    with pytest.raises(expected_exception=RecursionError):
        factorial(40000)

@pytest.mark.parametrize("n", [0.4, "hallo", True])
def test_factorial_type(n):
    with pytest.raises(expected_exception=TypeError) as exception_info:
        factorial(n)
    assert "int" in str(exception_info.value)  # Access exception info

@pytest.mark.parametrize("n", range(-1, -20, -2))
def test_factorial_negative_int(n):
    pytest.raises(ValueError, factorial, **{"n": n})  # Avoid this
```

temporary paths

```python
def test_temporary_paths(tmp_path_factory, tmp_path):
    t = tmp_path_factory.mktemp("extra_path")
    assert str(tmp_path).endswith("test_temporary_paths0")
```

What have we covered today?

Fixtures

in

Marks

How to handle errors?

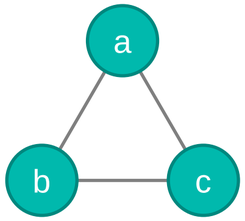
in pytest

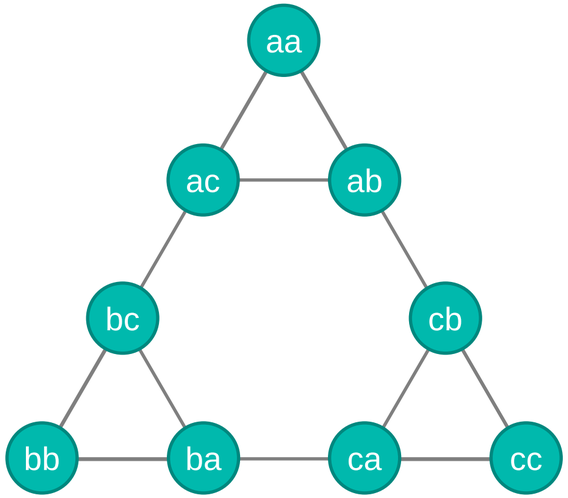
What have we covered today?

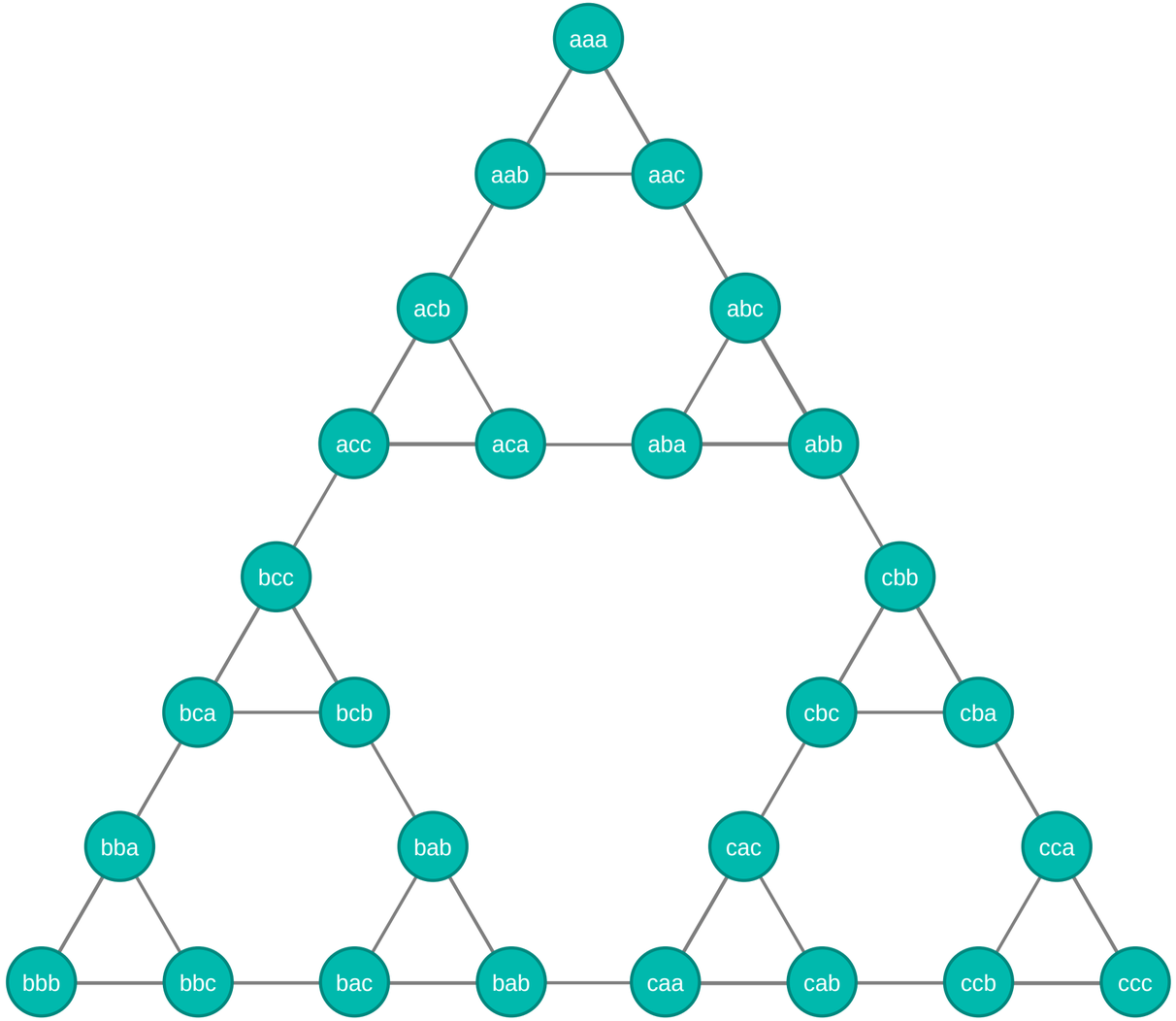
What about next week?

- What other testing platforms are there?
- Python build-in UnitTest VS Pytest
- How many tests should you write?
- How to make tests fast?
- Try test driven development
- How to integrate test in your CI/CD?
- Testing for Data Science

bigdata republic







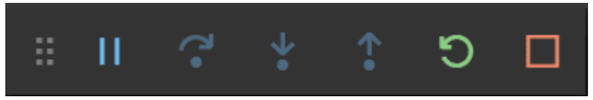
One last thing...

bigdata republic

We've all been there...

It will happen again...

A debugger, gets you here faster.



Phone +31 (0)168 479294 **Email** info@bigdatarepublic.nl **Address** Coltbaan 4C, 3439 NG Nieuwegein, The Netherlands