- Jedná se o nedílnou součást vývoje jakéhokoliv SW
- Některé typy testů je možné automatizovat
- Budeme se bavit o tzv. **unit testech** (jednotkové testy)
  - = každý testuje právě jednu věc
- Testování probíhá následujícím způsobem
  1) předáme vstup
  2) očekáváme daný výstup
  3) pokud výstup neodpovídá vstupu, test selže > [!info] > **test driven developement** … nejdříve napíšeme testy potom až kód
- V Pythonu máme k dispozici nativní možnost unit testů, avšak rozšířenější formou je balíček `pytest`

Balíček `pytest`

```
# soubor "test_inc.py"

def inc(x):
   return x + 1


# funkce test_inc provádějící test funkce inc
# assert již známe, pokud selže, selže i celkový test
def test_inc():
   assert inc(3) == 5
```

- Spuštěním příkazu `pytest` spustíme test

```
$ pytest
============================ test session starts ============================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 1 item

test_sample.py F                                                      [100%]

================================= FAILURES =================================
_____________________________ test_answer ______________________________

   def test_answer():
>       assert inc(3) == 5
E       assert 4 == 5
E        +  where 4 = inc(3)

test_sample.py:6: AssertionError
========================= short test summary info ==========================
FAILED test_sample.py::test_answer - assert 4 == 5
```

```
============================ 1 failed in 0.12s ============================
```

Organizace testů

- Především jde o strukturu v adresáři

```
setup.py
mypkg/
   __init__.py
   app.py
   view.py
tests/
   test_app.py
   test_view.py
   ...
```

- V případě větší možné kombinace vstupů a výstupů je vhodné použít *dekorátor* `@pytest.mark.parametrize` (test je parametrizovaný)

```python
import pytest

from algebra.vector import dot_product


@pytest.mark.parametrize(
   "v1, v2, result",
   [([1, 2, 3], [3, 2, 1], 10),
    ([-1, 2, 3], [3, 2, 1], 4)],
)
def test_dot_product(v1, v2, result):
   assert dot_product(v1, v2) == result
```

Fixtures

- Vidíme, že je nutné pro každý test vytvořit novou instanci třídy
- Použití tohoto dekorátoru nám zaručí, že se data pro každý test vytvoří znovu

```python
import pytest

from data.index import Index


@pytest.fixtures
def labels():
   return ["key 1", "key 2", "key 3", "key 4", "key 5"]
```

```python
@pytest.fixtures
def values():
   return [0, 1, 2, 3, 4]


# fixture může používat ostatní fixture
@pytest.fixtures
def index(labels):
   return Index(labels=labels)


# test používající dvě fixtures - index a labels
def test_index(index, labels):
   assert index.labels == test_labels
   assert isinstance(index.labels, list)
   assert index.name == ""


# test používající dvě fixtures - index a labels
def test_get_loc(index, values):
   assert values[index.get_loc("key 2")] == 1
```

Testování výjimek

```python
# testování situace kdy metoda musí vyvolat vyjimku
def test_invalid_key(index):
   with pytest.raises(KeyError):
       index.get_loc("key 10")
```

Testování v rámci docstringů

- Elegantní testy pro jednoduché funkce
- Příklady se rovněž zobrazí v nápovědě k funkci

```python
def sum_two_numbers(a, b):
   """Sums two numbers.

   Args:
       a: first number
       b: second number

   Example:
       >>> sum_two_numbers(10, 20)
       30
       >>> sum_two_numbers(-10, 20)
       10
   """
```

```
    return a + b
```

- Poté stačí přidat argument `--doctest-modules` při spuštění `pytest`

```
$ pytest --doctest-modules
======================== test session starts ========================
platform darwin -- Python 3.10.0, pytest-6.2.5, py-1.10.0, pluggy-1.0.0
rootdir: /Users/tomasmikula/Downloads/vyuka
plugins: cov-3.0.0
collected 1 item

sum_two_numbers.py .                                          [100%]

========================= 1 passed in 0.03s =========================
```

Code coverage

- Jedná se o procentuální pokrytí testů, neboli jaké procento zdrojového kódu je testováno
- V ideálním případě je dobré docílit 100% pokrytí, v praxi jsou však hodnoty nad 90% dostatečné

Jak počítat?

- V kombinaci s knihovnou `pytest` lze nainstalovat knihovnu `coverage`, která code coverage vypočítá

```
$ py -m pip install coverage
```

- Dále můžeme code coverage spočítat příkazem `coverage run -m pytest` (případně musíme dodat další argumenty jako `--doctest-modules`)

```
$ coverage run -m pytest

# Výsledek
$ coverage report -m
Name                      Stmts   Miss  Cover   Missing
-------------------------------------------------------
my_program.py                20      4    80%   33-35, 39
my_other_module.py           56      6    89%   17-23
-------------------------------------------------------
TOTAL                        76     10    87%
```