Wstęp do testowania w Pythonie

Mateusz Mazurek / m.mazurek@flomedia.pl

Spis treści

Wstęp

Cześć!

Ogromnie mi miło, że zdecydowałeś się zostać subskrybentem mojego newslettera. W zamian za okazane zaufanie, oddaję w Twoje ręce materiał, w którym poruszam jeden z ważniejszych elementów procesu dostarczania oprogramowania.

My, jako programiści, jesteśmy tylko ludźmi i zawsze będziemy popełniać błędy. W pełni tego świadomi, tworzymy właśnie testy. To właśnie one są głównym tematem tego dokumentu. O typach testów, ich przeznaczeniu i konkretnych sposobach użycia można by mówić wiele. W tym ebooku skupimy się na najbardziej popularnych typach testów - testach jednostkowych.

Po co tworzyć testy?

Zacznijmy najpierw od pewnej analogii. Na programowanie można patrzeć trochę jak na układanie puzzli. Z mniejszych elementów składamy większe. Niektóre puzzle utworzą nam koło samochodu, inny zbiór puzzli da nam drzwi auta, a jeszcze inny jego szybę. Te zbiory puzzli, połączone ze sobą, dadzą nam auto. Inne puzzle utworzą drzewa, jeszcze inne drogę i tak, po kolei, tworząc małe, pojedyncze elementy - budujemy cały obrazek.

Z programowaniem jest podobnie - aplikacje, które budujemy, składają się z małych elementów, takich jak pojedyncze funkcje czy klasy. Wraz z biegiem czasu będą one tworzyć znacznie bardziej zaawansowane komponenty, a na samym końcu - gotowe funkcjonalności.

Jest jednak jedna, zasadnicza różnica...

Jeden puzzel nie może być użyty w wielu miejscach obrazu. W programowaniu jest inaczej. Jako programiści, możemy, a nawet powinniśmy, stosować podejście "reuse". Czyli tworząc funkcjonalność B, korzystać z elementów funkcjonalności A. O ile układając puzzle od razu widzimy, że jakiś puzzel nie pasuje, to pisząc programy nie jest to już takie oczywiste. Na pierwszy rzut oka, element może pasować, ale później może się okazać, że nie jest to najlepsze dopasowanie.

Właśnie dzięki testom możemy wyłapywać takie sytuacje.

Ponadto testy pozwalają na zwiększenie:

- jakości tworzonej aplikacji, szczególnie wyłapywanie brzegowych przypadków,
- bezpieczeństwa podczas refactoringu,
- bezpieczeństwo podczas rozwiązywanie konfliktów w gicie

Ogólnie: sprawiają, że nasza aplikacja jest mniej zawodna!

PyTest

PyTest to tzw. "test runner", czyli framework, który definiuje sposób tworzenia testów, umie je wyszukiwać, wykonywać i rezultat pokazywać na ekranie. PyTest to główna biblioteka, którą się dziś zajmiemy.

Przykłady są pisane pod Pythonem w wersji 3.7. Prawdopodobnie będą działały w wyższych wersjach, ale niestety niższe mogą okazać się niekompatybilne.

Instalacja

Aby zainstalować tę bibliotekę należy skorzystać z managera pakietów, czyli z pip'a. Poniższe poleceniem uruchamiany w konsoli:

```
$ pip install pytest
```

Co powinno dać efekt podobny do

```
Collecting pytest
  Downloading pytest-5.4.3-py3-none-any.whl (248 kB)
     |▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒| 248 kB 1.1 MB/s
Collecting attrs>=17.4.0
  Downloading attrs-19.3.0-py2.py3-none-any.whl (39 kB)
Collecting pluggy<1.0,>=0.12
  Downloading pluggy-0.13.1-py2.py3-none-any.whl (18 kB)

...

  Downloading more_itertools-8.3.0-py3-none-any.whl (44 kB)
     |▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒| 44 kB 5.5 MB/s
Collecting py>=1.5.0
  Downloading py-1.8.1-py2.py3-none-any.whl (83 kB)
     |▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒| 83 kB 3.1 MB/s
Collecting six
  Downloading six-1.15.0-py2.py3-none-any.whl (10 kB)
Collecting pyparsing>=2.0.2
  Downloading pyparsing-2.4.7-py2.py3-none-any.whl (67 kB)
     |▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒| 67 kB 3.9 MB/s
Collecting zipp>=0.5
  Downloading zipp-3.1.0-py3-none-any.whl (4.9 kB)
Installing collected packages: attrs, zipp, importlib-metadata, pluggy, wcwidth, six,
pyparsing, packaging, more-itertools, py, pytest
Successfully installed attrs-19.3.0 importlib-metadata-1.6.1 more-itertools-8.3.0
packaging-20.4 pluggy-0.13.1 py-1.8.1 pyparsing-2.4.7 pytest-5.4.3 six-1.15.0 wcwidth-
0.2.4 zipp-3.1.0
```

Pierwszy test

Stwórzmy plik *main1.py* z zawartością:

```
def sub(a: int, b: int) -> int:
    return a - b
```

i drugi plik o nazwie *test_main1.py* z zawartością:

```
from main1 import sub


def test_sub():
    first_arg = 2
    second_arg = 6

    expected = -4

    assert sub(first_arg, second_arg) == expected
```

Zanim przejdziemy do uruchomienia, zauważ co zrobiliśmy. Napisaliśmy funkcję, która odejmuje od siebie liczby oraz test, który sprawdza, czy wynik odejmowania jest poprawny. Teraz możemy uruchomić test, korzystając z polecenia `pytest`:

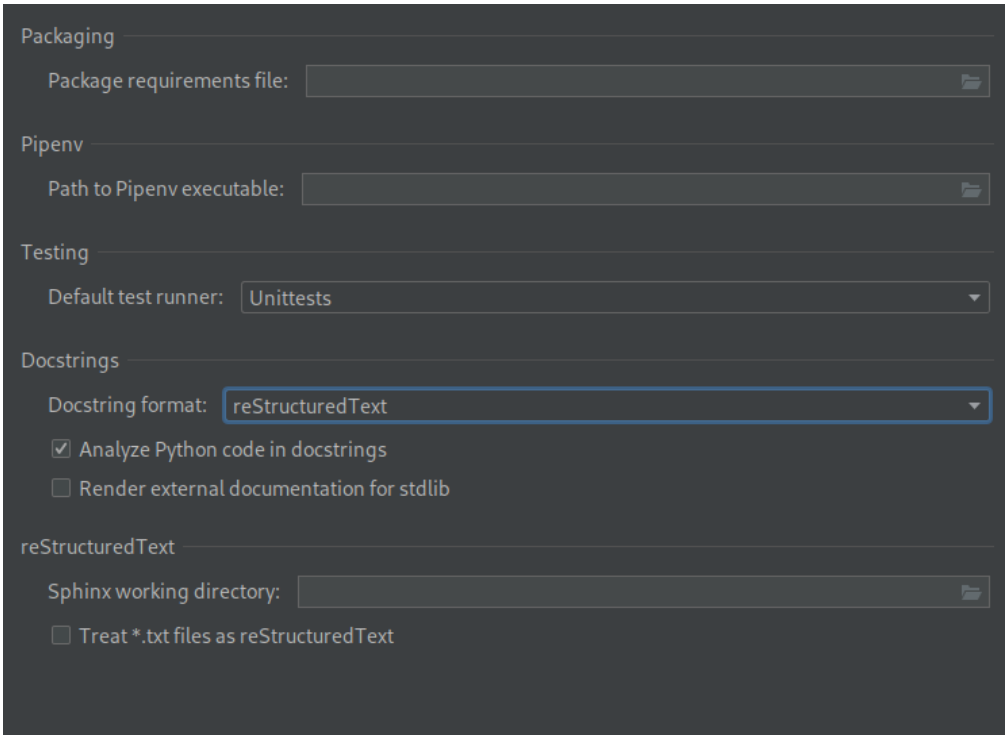
```
$ pytest test_main1.py
================= test session starts ==================
platform linux -- Python 3.7.7, pytest-5.4.3, py-1.8.1, pluggy-0.13.1
rootdir: /home/mmazurek/testymm2
collected 1 item

test_main1.py [100%]

================== 1 passed in 0.01s ===================
```

W rezultacie otrzymaliśmy informację że jeden test przeszedł! Gratuluję Ci :)

Konfiguracja w PyCharmie

Testy najwygodniej uruchamia się z poziomu PyCharm'a. Żeby móc to robić trzeba go skonfigurować. Stwórz proszę nowy projekt i w nim dwa pliki, takie jak w poprzednim przykładzie.

Po lewej na górze wejdź w menu **File** → **Settings**, następnie z listy po lewej wybierz **Tools** i pod nim wskaż **Python Integrated Tools**. Po kliknięciu w ten element po prawej stronie zobaczysz:

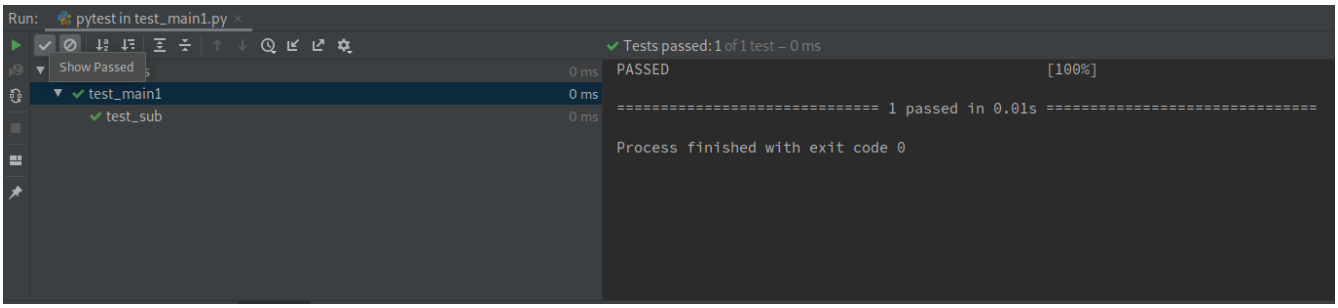


W polu "Default test runner" wskaż "pytest" i kliknij OK.

I już, PyCharm umie uruchamiać testy korzystając z PyTesta!

Uruchomienie z poziomu PyCharm'a

Aby uruchomić test z poziomu Pycharma, kliknij na plik z testem i naciśnij "Run 'pytest in test_main1.py...'". Spowoduje to uruchomienie testów:



Zauważ, że PyCharm za pliki z testami będzie uważał pliki rozpoczynające się od "test_" lub kończące się na "_test". Metody w tych plikach, które są testami powinny zaczynać się od "test".

Przykład

Mamy kawałek kodu w pliku *main2.py*:

```
def is_palindrome(n):
    n = str(n).replace(' ', '').lower()
    return n == n[::-1]
```

Sprawdza on, czy przekazany string jest palindromem, dokonując niezbędnych formatowań, żeby takie rzeczy jak spacje czy wielkość liter nie zepsuły wyniku.

I teraz plik *test_main2.py*:

```
from main2 import is_palindrome


def test_is_palindrome():
    assert is_palindrome("atak kata")
    assert is_palindrome("ada bzy zbada")


def test_is_not_palindrome():
    assert not is_palindrome("mama")
    assert not is_palindrome("wiedzieć")
    assert not is_palindrome("mmazurek.dev")
```

Możemy takie testy sparametryzować, tzn. zdefiniować "przykłady", które test powinien sprawdzić. Wtedy test mógłby wyglądać tak:

```
from main2 import is_palindrome
import pytest


@pytest.mark.parametrize("n, result", [("atak kata", True), ("ada bzy zbada", True),
("mama", False), ("wiedzieć", False), ("mmazurek.dev", False)])
def test_is_palindrome(n, result):
    assert is_palindrome(n) == result
```

Używając dekoratora `pytest.mark.parametrize` stworzyliśmy test, który testuje kilka przypadków. Podaliśmy wartość przekazywaną do funkcji oraz oczekiwany rezultat funkcji. Znacznie wygodniej tak pisać testy niż robić to ręcznie. PyCharm przy uruchomieniu ładnie pokazuje wynik:

```
platform linux -- Python 3.7.7, pytest-5.4.3, py-1.8.1, pluggy-0.13.1 --
/home/mmazurek/testymm2/venv/bin/python
cachedir: .pytest_cache
rootdir: /home/mmazurek/testymm2
collecting ... collected 5 items

test_main2.py::test_is_palindrome[atak kata-True] PASSED                 [ 20%]
test_main2.py::test_is_palindrome[ada bzy zbada-True] PASSED             [ 40%]
test_main2.py::test_is_palindrome[mama-False] PASSED                     [ 60%]
test_main2.py::test_is_palindrome[wiedzie\u0107-False] PASSED            [ 80%]
test_main2.py::test_is_palindrome[mmazurek.dev-False] PASSED             [100%]
```

Przykład z wyjątkami

Czasem zdarza się, że oczekiwanym zachowaniem testowanej metody jest rzucenie wyjątku, np. tutaj:

```
def is_palindrome(n):
    if not isinstance(n, str):
        raise TypeError("Invalid argument. Supporting only strings.")
    n = str(n).replace(' ', '').lower()
    return n == n[::-1]
```

Jak takie coś przetestować? PyTest pozwala łatwo sobie z tym poradzić. Korzystając z `pytest.raises` możemy przetestować to tak:

```
from main3 import is_palindrome
import pytest


def test_is_palindrome_invalid_arg():
    with pytest.raises(TypeError):
        is_palindrome(5)
        is_palindrome(10)
        is_palindrome(5.3)
        is_palindrome(None)
```

Podsumowanie

Dzięki temu, że poświęciłeś czas na przeczytanie tego dokumentu, nabyłeś nową wiedzę z zakresu podstaw testowania w Pythonie. Przerobiliśmy:

- instalację PyTesta,
- napisanie i uruchomienie pierwszego testu,
- konfigurację PyCharma,
- przykład z palindromem,
- przykład z parametryzacją,
- przykład z wyjątkami.

Mam nadzieję, że wiedza okazała się przydatna. Jeśli pytasz co dalej, to naturalnym, kolejnym krokiem w tym temacie, byłoby poznanie atrap, czyli sposobu na radzenie sobie z zależnościami testowanych elementów.

Zostawiam również dwa linki do mojego bloga:

- Testowanie w duchu BDD,
- Testy Selenium,

które pokazują trochę inne podejście do testowania.

Dzięki za poświęcony czas.

Pozdrawiam,