LABORATORIUM NR 13:

Piotr Suchy 407332

Pierwszy test - hello:

Faza RED:

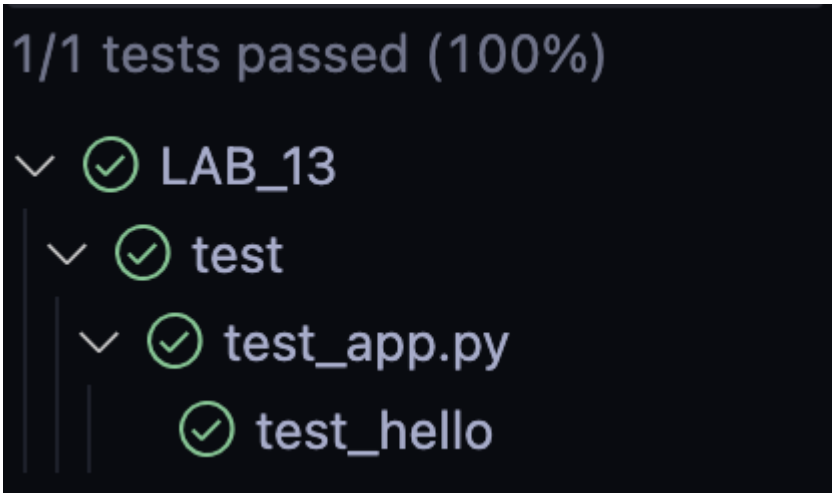
```
test/test_app.py FFFFF                                                   [100%]

==================================== FAILURES ====================================
_____________________________________ test_hello _____________________________________

    def test_hello():
        got = hello("Aleksandra")
        want = "Hello Aleksandra"

>       assert got == want
E       AssertionError: assert None == 'Hello Aleksandra'
```

FAZA GREEN:

Treść testu:

```
def test_hello():
    got = hello("Aleksandra")
    want = "Hello Aleksandra"

    assert got == want
```

Wynik testu po napisaniu funkcji hello():



FAZA REFACTOR:

Ponieważ funkcja hello() jest bardzo prymitywna, faza refactor była zbędna w tym przypadku.

Drugi test - Extract Sentiment:

Faza RED:

```
test/test_app.py:11: AssertionError
_____________ test_extract_sentiment[I think today will be a great day] _____________

sample = 'I think today will be a great day'

    @pytest.mark.parametrize('sample', testdata)
    def test_extract_sentiment(sample):

        sentiment = extract_sentiment(sample)

>       assert sentiment > 0
E       TypeError: '>' not supported between instances of 'NoneType' and 'int'
```

FAZA GREEN:

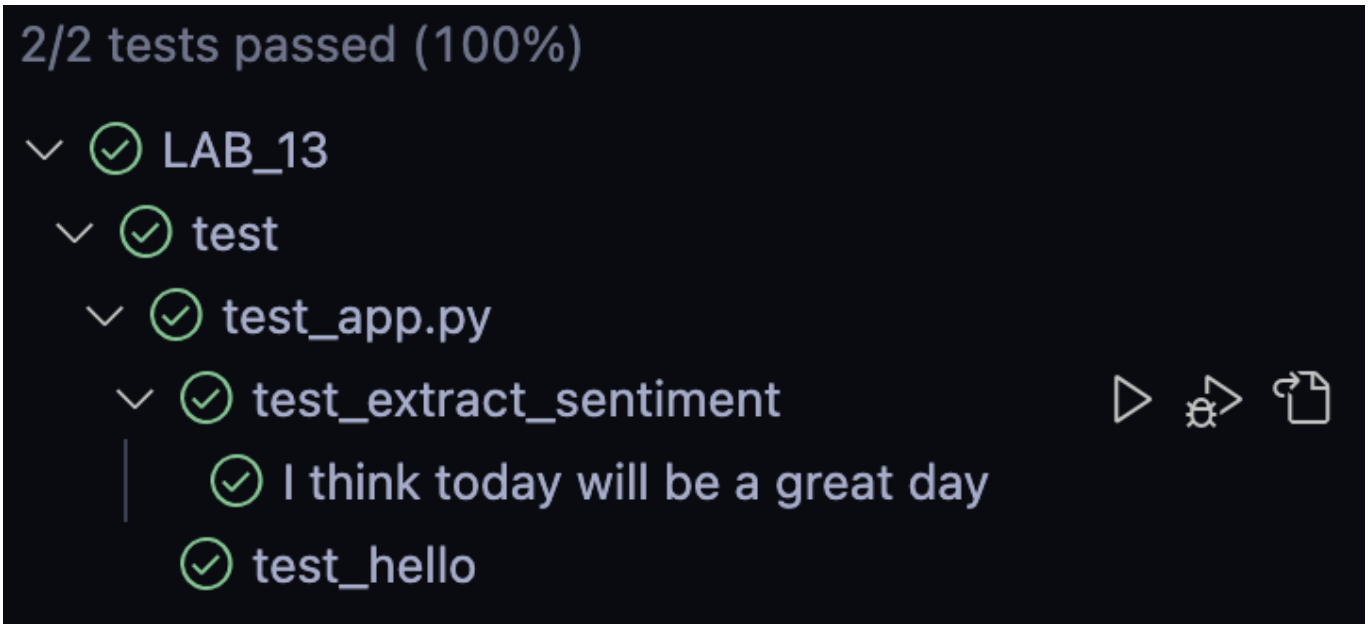
Treść testu:

```
testdata = ["I think today will be a great day"]
@pytest.mark.parametrize("sample, expected", testdata)
def test_extract_sentiment(sample):

    sentiment = extract_sentiment(sample)

    assert sentiment > 0
```

Treść funkcji:

```
6   def extract_sentiment(text):
7       text = TextBlob(text)
8       return text.sentiment.polarity
```

Wynik testu po napisaniu funkcji extract_sentiment():



FAZA REFACTOR:

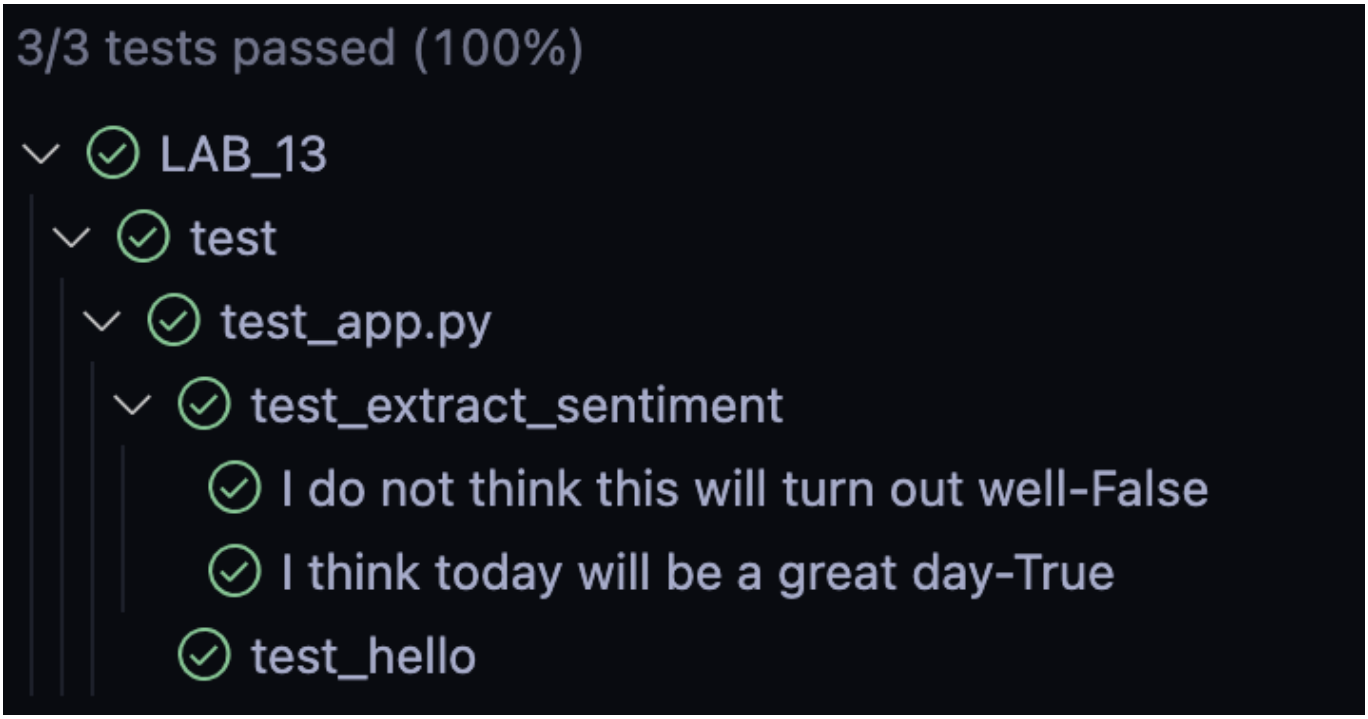
Refactoring: Zmieniona treść testu:

```
testdata = [("I think today will be a great day", True), ("I do not think this will turn out well", False)]

@pytest.mark.parametrize("sample, expected", testdata)
def test_extract_sentiment(sample, expected):

    sentiment = extract_sentiment(sample)

    assert sentiment == expected
```

Zmieniona treść funkcji:

```
6   def extract_sentiment(text):
7       text = TextBlob(text)
8       if text.sentiment.polarity > 0:
9           return True
10      else:
11          return False
```

Wyniki po refactoringu:



Trzeci test - text function:

Faza RED:

```
test/test_app.py:21: TypeError
_________ test_text_contain_word[There is a duck in this text-duck-True] _________

sample = 'There is a duck in this text', word = 'duck', expected_output = True

    @pytest.mark.parametrize('sample, word, expected_output', testdata)
    def test_text_contain_word(sample, word, expected_output):

>       assert text_contain_word(word, sample) == expected_output
E       AssertionError: assert None == True
E        +  where None = text_contain_word('duck', 'There is a duck in this text')
```

```
test/test_app.py:32: AssertionError
____________ test_text_contain_word[There is nothing here-duck-False] ____________

sample = 'There is nothing here', word = 'duck', expected_output = False

    @pytest.mark.parametrize('sample, word, expected_output', testdata)
    def test_text_contain_word(sample, word, expected_output):

>       assert text_contain_word(word, sample) == expected_output
E       AssertionError: assert None == False
E        +  where None = text_contain_word('duck', 'There is nothing here')
```

FAZA GREEN:

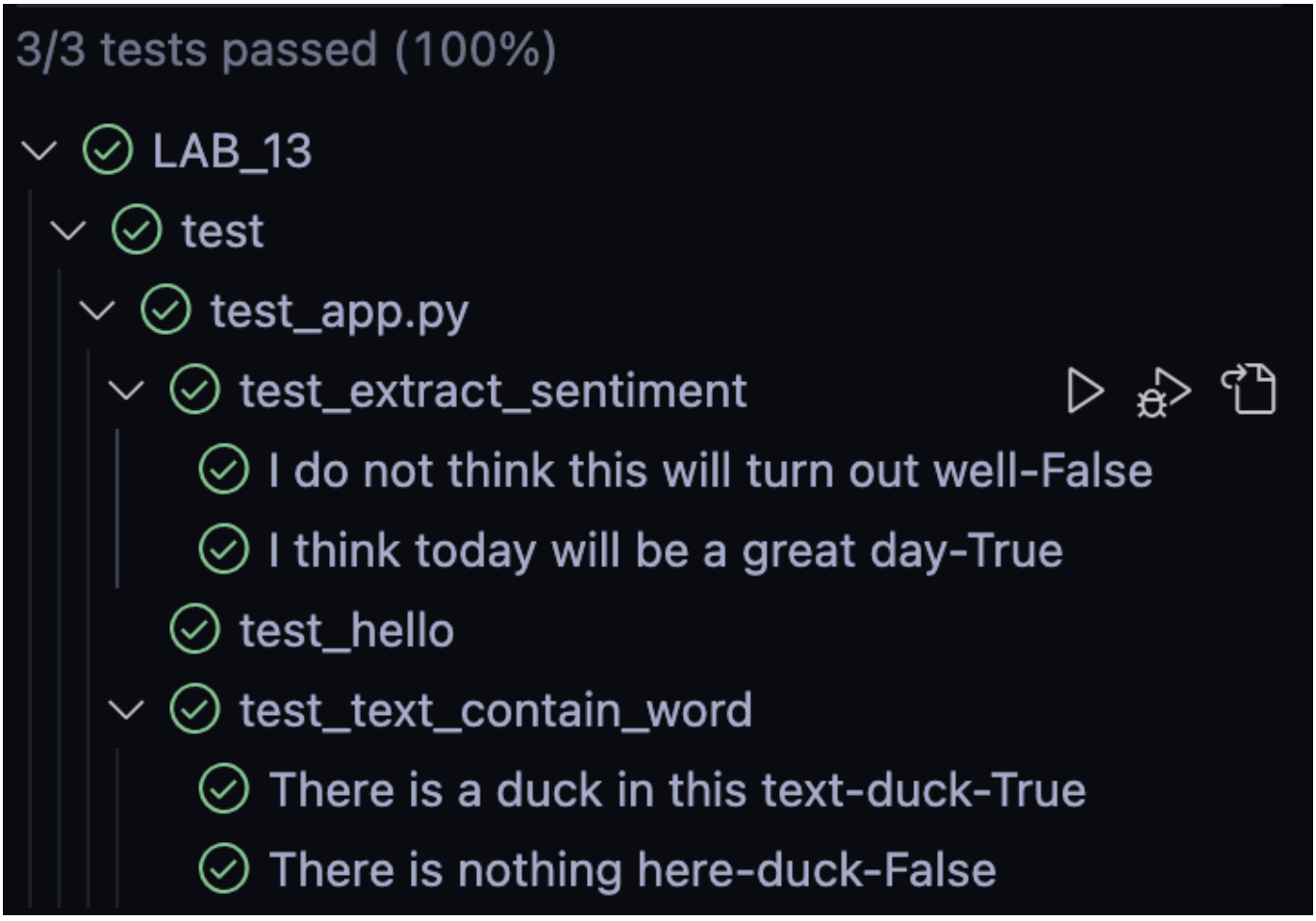
Treść funkcji:

```python
def text_contain_word(word: str, text: str):
    return word in text
```

Treść testu:

```python
testdata = [
    ('There is a duck in this text', 'duck', True),
    ('There is nothing here', 'duck', False)
    ]

@pytest.mark.parametrize('sample, word, expected_output', testdata)
def test_text_contain_word(sample, word, expected_output):

    assert text_contain_word(word, sample) == expected_output

```

Wynik testu po napisaniu funkcji:



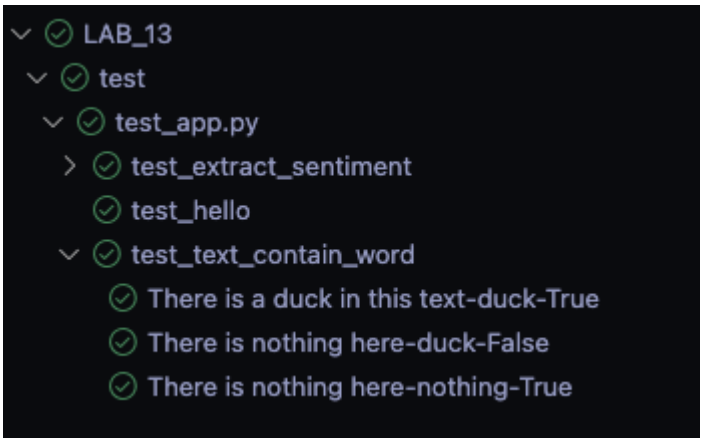
FAZA REFACTOR:

Ponieważ funkcja text_contain_word() jest jednolinijkowa, nie ma przestrzeni do ulepszeń. Ale można dodać jeszcze jeden przypadek testu, aby mieć więcej pewności.

Treść funkcji testującej po refactorze:

```
testdata = [
    ('There is a duck in this text', 'duck', True),
    ('There is nothing here', 'duck', False),
    ('There is nothing here', 'nothing', True)
    ]

@pytest.mark.parametrize('sample, word, expected_output', testdata)
def test_text_contain_word(sample, word, expected_output):

    assert text_contain_word(word, sample) == expected_output
```

Wynik testów po refactorze:



Czwarty test - Przykład dla bubblesort'u:

Faza RED:

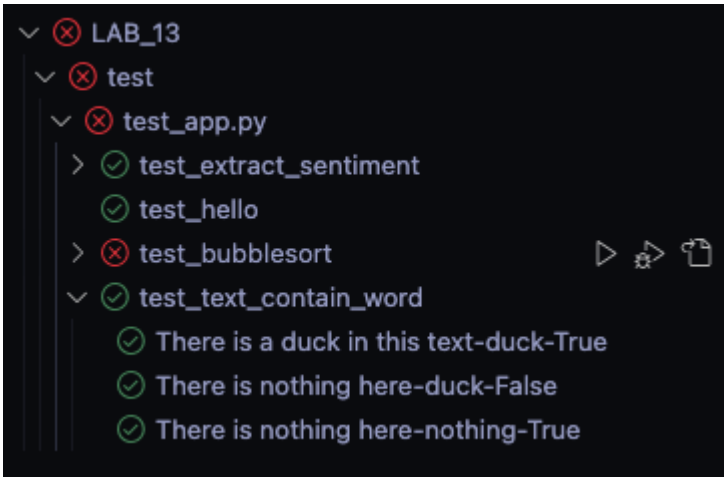
Treść testu:

```
def test_bubblesort():
    testdata = [[6, 0, 3, 1, 2, 5, 6, 7, 2, 4, 5]]

    assert bubblesort(testdata) == testdata.sort()
```

Treść funkcji - póki co nie robi nic, oprócz zwrócenia parametru:

```
19    def bubblesort(list_of_numbers: list):
20        return list_of_numbers
21
```

I wynik testu dla funkcji, która nic nie robi (póki co) - jak widać test niezaliczony:



Faza Green:

Treść funkcji, która przechodzi test:

```
def bubblesort(list_of_numbers: list):
    list_to_sort = list_of_numbers
    n_of_swaps = 1
    i = 0
    while n_of_swaps != 0:
        n_of_swaps = 0
        for i in range(len(list_to_sort)-1):
            if list_to_sort[i] > list_to_sort[i+1]:
                n_of_swaps += 1
                list_to_sort[i], list_to_sort[i+1] = list_to_sort[i+1], list_to_sort[i]
                i += 1

    return list_to_sort
```

Treść testu, przed refactoringiem:

```
def test_bubblesort():
    testdata = [6, 0, 3, 1, 2, 5, 6, 7, 2, 4, 5]
    assert bubblesort(testdata) == sorted(testdata)
```

Wynik testu dla funkcji powyżej:

Faza Refactor:

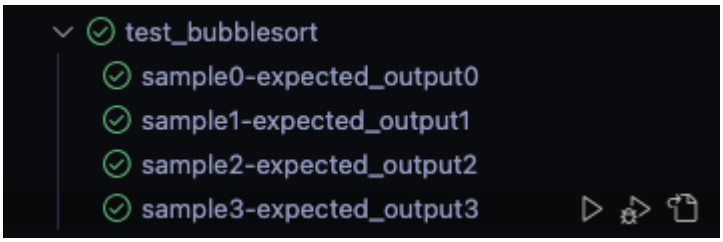
Po dodaniu dodatkowych testów - edge casów:

```
testdata = [
    ([0, 2, 3, 1, 5, 6, 7, 4, 5, 3], sorted([0, 2, 3, 1, 5, 6, 7, 4, 5, 3])),
    ([-1, 2, -4, -3, 0, 0, -2, 3, 4], sorted([-1, 2, -4, -3, 0, 0, -2, 3, 4])),
    ([1], [1]),
    ([], [])
    ]

@pytest.mark.parametrize('sample, expected_output', testdata)
def test_bubblesort(sample, expected_output):

    assert bubblesort(sample) == expected_output
```

I zmodyfikowaniu funkcji do rozpatrywania edge case'a + usunięcia zbędnego kodu:

```
def bubblesort(list_of_numbers: list):
    n = len(list_of_numbers)
    if n == 0:
        return []
    for i in range(n):
        swap_occured = False
        for j in range(0, n-i-1):
            if list_of_numbers[j] > list_of_numbers[j+1]:
                list_of_numbers[j], list_of_numbers[j+1] = list_of_numbers[j+1], list_of_numbers[j]
                swap_occured = True
        if not swap_occured:
            return list_of_numbers

```

Dostajemy wyniki:



WNIOSKI:

TDD jest przydatnym sposobem tworzenia kodu, szczególnie w większych projektach. TDD sprawia, że kod, który stworzymy od razu spełnia wszystkie wymogi. Podejście to jest jednak dosyć powolne i w indywidualnych projektach nie jest wymagane.