KI/TSW Sem2

Pavel Beranek

February 25, 2025

KI/TSW 2025

Seminář 2 - Testovací případy

Zadání

Cílem je otestovat jednoduchou softwarovou kalkulačku pomocí testových případů s využitím dvou rozlišných test runnerů, navrhnout další testové případy pomocí analýzy hraničních hodnot a ovládat jejich spuštění pomocí značkování a parametrizace.

Vytvořte si nějaký projekt a vložte do nového souboru tento kód v jazyce Python, který bude naším testovaným objektem.

```
# calculator.py
class Calculator:
    def add(self, a, b):
        return a + b

    def subtract(self, a, b):
        return a - b

    def multiply(self, a, b):
        return a * b

    def divide(self, a, b):
        if b == 0:
            raise ValueError("Cannot divide by zero")
        return a / b
```

Úkol 1 - Tvorba testových případů v UnitTest

Modul unittest je modul ze standardní knihovny jazyka Python a není nic nutné instalovat. Spusťte si tyto existující testy pomocí příkazu: `python -m unittest test_calculator.py`

```
#test_calculator.py
import unittest
from calculator import Calculator

class TestCalculator(unittest.TestCase): # testovací třída
    def setUp(self):
        self.calc = Calculator()

    def test_add(self): # testovací funkce/metoda
        self.assertEqual(self.calc.add(2, 3), 5) # testovací případ
        self.assertEqual(self.calc.add(-1, 1), 0)

    def test_subtract(self):
        self.assertEqual(self.calc.subtract(5, 2), 3)
        self.assertEqual(self.calc.subtract(2, 5), -3)

    def test_multiply(self):
        self.assertEqual(self.calc.multiply(3, 4), 12)
        self.assertEqual(self.calc.multiply(0, 5), 0)

    def test_divide(self):
        self.assertEqual(self.calc.divide(10, 2), 5)
        self.assertRaises(ValueError, self.calc.divide, 10, 0)

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

V testové třídě vidíme jednotlivé testové případy v testových metodách/funkcích. Všechny metody potřebují ke své funkčnosti existenci instance kalkulačky, tzv. Fixture, který se vytváří metodou setUp. Existuje i opačný proces odstraňování závislostních prvků, který se nazývá tearDown.

Obecná teorie testování nezná termíny jako testovací třída nebo testovací funkce/metoda. Zná pouze pojem testovací případ (test case), který představuje atom manuálního i automatizovaného testování. Testovací třídy a metody nám slouží pro organizaci testových případů. Vyšší obecná úroveň jsou testovací scénáře, které představují konkrétní varianty využívání aplikace klientem, které lze občas vyjádřit i v kódu, ale ne v našem primitivním příkladě kalkulačky.

Pokud spuštění funguje, tak přidejte ke každé testovací metodě alespoň jeden testovací případ.

Úkol 2 - Organizace testů pomocí AAA pracovního rámce

Pro lepší udržitelnost a čitelnost testovacího kódu se obyčejně testy strukturují pomocí AAA pracovního rámce. AAA pracovní rámec dělí testy na tři fáze:

1. Arrange (příprava) – inicializace testovaných objektů a nastavení vstupních hodnot.
2. Act (akce) – volání testované metody/funkce.
3. Assert (ověření výsledku) – kontrola výstupu proti očekávané hodnotě.

Příklad testovacího případu strukturovaného pomocí AAA:

```
class TestCalculator(unittest.TestCase): # testovací třída
    def setUp(self):
        self.calc = Calculator()

    def test_add_two_positive_integers(self): # testovací metoda teď reprezentuje test-case
        # ARRANGE: Příprava vstupních dat
        a = 2
        b = 3
        expected_result = 5

        # ACT: Spuštění testované funkce
        result = self.calc.add(a, b)

        # ASSERT: Ověření výsledku
        self.assertEqual(result, expected_result)
```

Z důvodu navýšení počtu metod je možné organizovat testy nad objektem do více souborů.

- test_addition.py
- test_subtraction.py
- test_multiplication.py
- test_division.py

Přepište testovací případy pomocí AAA pracovního rámce. Reorganizaci projektu nechám na Vás. Není to vyloženě nutné.

Úkol 3 - Přepsání testových případů do Pytest

Pytest je jeden z nejoblíbenějších spouštěčů testů (tzv. test runner). Nainstalujte si modul pytest pomocí: `python -m pip install pytest`.

Vyzkoušejte si spuštění následujícího testwaru:

```
import pytest
from calculator import Calculator

@pytest.fixture
def calc():
    return Calculator()

def test_add(calc):
    assert calc.add(2, 3) == 5
    assert calc.add(-1, 1) == 0

def test_subtract(calc):
    assert calc.subtract(5, 2) == 3
    assert calc.subtract(2, 5) == -3

def test_multiply(calc):
    assert calc.multiply(3, 4) == 12
    assert calc.multiply(0, 5) == 0

def test_divide(calc):
    assert calc.divide(10, 2) == 5
    with pytest.raises(ValueError):
        calc.divide(10, 0)
```

Přepište Vaše navíc vytvořené testové případy v UnitTestu do Pytestu. Testovací případy budou funkce a můžete je rozdělit do souborů (test_násobení, test_sčítání atd.) nebo je nechat v jednom velkém. Implementujte testy pomocí AAA pracovního rámce.

Úkol 4 - Parametrizace testů

Pokud by Vás nebavilo psát dokola velké množství kódu pro hraniční hodnoty, pak je možné využít techniku parametrizace v Pytest. Dojde k velké úspoře místa za cenu zmenšení přehlednosti kódu. Přehlednost kódu lze dodat do parametrizovaných testů dodáním identifikátoru.

```
@pytest.mark.parametrize("a, b, expected", [
    pytest.param(1, 2, 3, id="positive_numbers"),
    pytest.param(-1, -1, -2, id="negative_numbers"),
    pytest.param(0, 0, 0, id="zero_case")
])
def test_add(calc, a, b, expected):
    assert calc.add(a, b) == expected
```

Vyzkoušejte si do kódu dodat parametrizované testy a udělejte si závěr o způsobu testování, který Vám přijde nejpřehlednější. Budete si muset poradit nějak vhodně s výjimkou.

Úkol 5 - Implementace testů hraničních hodnot

Testy navrhujete tak, abyste otestovali očekávané chování nad všemi domény ekvivalence. Doména ekvivalence je sada hodnot, pro které se systém chová stejně. Tyto domény mají své hranice, takže stačí testovat hodnoty na hranicích a kolem hranic.

Příklad:

```
def is_valid_age(age):
    return 18 <= age <= 65
```

Typy vstupů:

- Platné ekvivalentní vstupy: 20, 40, 60
- Neplatné ekvivalentní vstupy: 10, 70
- Hraniční hodnoty minimální hranice: 17, 18, 19
- Hraniční hodnoty maximální hranice: 64, 65, 66

Doporučené testovací případy:

```
assert is_valid_age(17) == False  # těsně pod hranicí
assert is_valid_age(18) == True   # přesně na spodní hranici
assert is_valid_age(19) == True   # těsně nad hranicí
assert is_valid_age(64) == True   # těsně pod horní hranicí
assert is_valid_age(65) == True   # přesně na horní hranici
assert is_valid_age(66) == False  # těsně nad hranicí
```

V případě kalkulačky to budou primárně extrémní hodnoty, zadání s nulou, nevhodné datové typy atd. Příklady hraničních hodnot pro naší kalkulačku:

- Operace s 0 (násobení, dělení).
- Operace s velmi malými a velkými čísly (1e-10, 1e10).
- Nejmenší a největší hodnoty v int a float rozsahu (-2^31, 2^31-1, -inf, inf).
- Operace s jinými datovými typy než číselnými.

Tyto navržené testové případy implementujte a spusťte.

Úkol 6 - Selektivní testování pomocí značek

Selektivní spouštění vybraných skupin testů lze provádět v Pytest několika způsoby:

1. Spuštění konkrétního souboru
2. Spuštění konkrétní testovací třídy nebo metody
3. Spuštění filtrováním podle názvu
4. Spuštění filtrováním pomocí značek
5. Spuštění filtrováním podle výsledků testů (neúspěšné, změněné, N nejpomalejších)

Nejprve v projektovém adresáři vytvořte soubor `pytest.ini` a dodejte vlastní značky. Příklad struktury zápisu:

```
[pytest]
markers =
    addition: Testy pro sčítání
    multiplication: Testy pro násobení
```

V kódu se testovací případy označí definovanou značkou:

```
@pytest.mark.multiplication
def test_multiply(calc):
    assert calc.multiply(3, 4) == 12
```

Selektivní výběr testů se pak provádí pomocí např. `pytest -m multiplication`.

Vyzkoušejte si přidat selektivní testy pomocí značek do Vašeho testovacího kódu a selektivně testy spustit. Vytvořte značky pro jednoduché případy, testy extrémních hodnot a výjimkové testy (pokud jste minulý úkol řešili oddělením výjimek z parametrizovaného testu pro dělení)..

Úkol 7 - Měření pokrytí příkazů a větví

Pytest generuje pro uživatele velké množství výstupu (je nutné doinstalovat pluginy) pro testovací reporty. Při spuštění Pytest bez pluginů se objeví v terminálu výstupy testů se značkou o výsledku:

- . = úspěšný test
- F = neúspěšný test
- E = chyba při běhu testu
- Počet testů a celkový čas běhu

Kromě samotného výpisu o úspěšnosti testů je vhodné vypisovat důležité testovací metriky. Mezi nejdůležitější metriky patří metriky pokrytí (coverage). Výpočet metrik pokrytí je nutný doinstalovat `pip install pytest-cov`. Po instalaci je možné měřit 2 metriky pokrytí:

1. Pokrytí výrazů (statement coverage, SC): `pytest --cov=calculator`
2. Pokrytí větví (branch coverage, BC): `pytest --cov=calculator --cov-branch`

Vysvětlení si provedeme na následujícím kódu:

```
def classify_number(n):
    if n > 0:
        if n % 2 == 0:
            return "Positive Even"
        else:
            return "Positive Odd"
    else:
        return "Not Positive"
assert classify_number(2) == "Positive Even"
assert classify_number(-1) == "Not Positive"
```

Výsledek bude SC=100 % (prošli jsme každým uzlem, vyznačeno kulatou závorkou) a BC != 100 % (neprošli jsme každou cestou, vyznačeno lomítkama).

```
        (n > 0?)
         /    \
       Ano    "Not Positive"
      /     \
 (n % 2 == 0?)
   /          \
"Even"      "Odd"
```

Nainstalujte modul pro měření pokrytí: pip install pytest-cov a vypište si Vaše pokrytí. V takto jednoduchém kódu byste měli mít 100 % u obou metrik. vygenerujte si do HTML report a prohlédněte si metriky SC a BC.

Úkol 8 - Generování testovacího reportu

Závěr testování z pohledu testera je tvorba reportu o testování. Šablony pro testovací reporty Vám dodá testovací manager, avšak můžete si vygenerovat primitivní automatické v různých formátech. V Pytest je možné vygenerovat jednoduchý export do HTML nebo XML/JSON pro další parsování na různých úrovních detailnosti (je nutné doinstalovat pluginy do pytest):

- HTML:
  - pip install pytest-html
  - pytest –html=report.html
- XML:
  - pytest –junitxml=report.xml
- JSON:
  - pip install pytest-cov
  - pytest –cov-report json

Řešení

Předložené řešení není úplné, ale obsahuje všechny technikálie nutné pro řešení.

```
import pytest
from calculator import Calculator


#selektivni spusteni testu: pytest -m standard_arithmetic_tests
#report do html: pytest --cov=calculator --cov-branch --htmk=test_report.html


@pytest.fixture
def calc():
    return Calculator()


@pytest.mark.standard_arithmetic_tests
@pytest.mark.parametrize("a, b, expected_result", [
    pytest.param(2, 3, 5, id='positive_sum_case'),
    pytest.param(-1, 1, 0, id='sum_to_zero'),
    pytest.param(-1, -5, -6, id='negative_sum_case'),
])
def test_add(calc, a, b, expected_result):
    assert calc.add(a, b) == expected_result


@pytest.mark.extreme_arithmetic_tests
@pytest.mark.parametrize('a, b, expected_result', [
    pytest.param(1E-10, 1E-10, 2E-10, id='sum_extreme_low_values'),
    pytest.param(1E10, 1E10, 2E10, id='sum_extreme_high_values'),
])
def test_extreme_add(calc, a, b, expected_result):
    assert calc.add(a, b) == expected_result


@pytest.mark.standard_arithmetic_tests
@pytest.mark.parametrize('a, b, expected_result', [
    pytest.param(5, 2, 3, id='positive_sub_result'),
    pytest.param(2, 5, -3, id='negative_sub_result'),
    pytest.param(-3, -3, 0, id='sub_two_negatives'),
])
def test_sub(calc, a, b, expected_result):
    assert calc.subtract(a, b) == expected_result

@pytest.mark.extreme_arithmetic_tests
@pytest.mark.parametrize('a, b, expected_result', [
    pytest.param(1E-10, 1E-10, 0, id='sub_extreme_low_values'),
    pytest.param(1E10, 1E10, 0, id='sub_extreme_high_values')
])
def test_extreme_sub(calc, a, b, expected_result):
    assert calc.subtract(a, b) == expected_result


@pytest.mark.standard_arithmetic_tests
@pytest.mark.parametrize('a, b, expected_result', [
    pytest.param(3, 4, 12, id='mul_two_positives'),
```

```
    pytest.param(0, 5, 0, id='mul_with_zero'),
    pytest.param(-2, 5, -10, id='mul_by_negative'),
])
def test_mul(calc, a, b, expected_result):
    assert calc.multiply(a, b) == expected_result


@pytest.mark.standard_arithmetic_tests
@pytest.mark.parametrize('a, b, expected_result', [
    pytest.param(10, 2, 5, id='divide positive by positive'),
    pytest.param(4, -2, -2, id='divide positive by negative')
])
def test_div(calc, a, b, expected_result):
    assert calc.divide(a, b) == expected_result

@pytest.mark.exception_tests
def test_division_by_zero(calc):
    with pytest.raises(ValueError):
        calc.divide(5, 0)
```

Domácí cvičení

1. Naprogramujte funkci is_valid_password(password: str) -> bool, která splňuje následující pravidla:

- Heslo musí mít minimálně 8 znaků.
- Musí obsahovat alespoň jedno číslo.
- Musí obsahovat alespoň jedno velké písmeno.
- Musí obsahovat alespoň jedno malé písmeno.
- Pokud heslo nesplňuje podmínky, funkce vrátí False.

2. Navrhněte alespoň 6 testovacích případů, které pokryjí různé vstupy:

- Korektní heslo
- Příliš krátké heslo
- Heslo bez čísla
- Heslo bez velkého písmena
- Heslo bez malého písmena
- Heslo obsahující jen čísla

3. Napište testy v pytest s použitím parametrizace.

4. Vytvořte report z testování.