České vysoké učení technické v Praze FIT

Programování v Pythonu

Jiří Znamenáček

Příprava studijního programu Informatika je podporována projektem financovaným z Evropského sociálního fondu a rozpočtu hlavního města Prahy.

Praha & EU: Investujeme do vaší budoucnosti

Python – Unit-testy

Úvod

Úkolem „testů jednotek“ (*unit test*) je automaticky a zcela samostatně otestovat (pokud možno) každou část Vašeho kódu, zda se chová podle předpokladů. A to kdykoliv během vývoje kódu, dokonce třeba i úplně na začátku, když váš kód nedělá ještě vůbec nic! (Testy v tu chvíli *musí* selhat a takovému přístupu k psaní programů se říká *vývoj řízený testy*, angl. *TDD, test-driven development*.) Typicky se *unit testing* spojuje dohromady s *code coverage*.

Kromě toho, že vám testy pomohou zkontrolovat funkčnost vašeho kódu, pomáhají vám též v jeho lepším rozvržení (musíte si dopředu promyslet, co a jak přesně by vlastně váš program měl dělat, asi o něco důkladněji než normálně) a v neposlední řadě usnadňují jeho spolupráci s kódem cizím.

Testování v Python'u

V Python'u máme k dispozici testování několikerého druhu, a to jak přímo uvnitř standardní distribuce jazyka, tak z několik externích míst:

- `unittest` – modul určený přímo na testování jednotek; filozoficky (i autorstvím) patří do rodiny testovacích nástrojů *xUnit* a je trošku „ukecanější“ než jiné frameworky, ale je k dispozici v každé instalaci Python'u
- `doctest` – speciální pythoní modul určený k testování interaktivních příkladů z dokumentačních řetězců (ano, dokonce i ty je možné testovat)
- externí nástroje pro testování jako třeba *nose* nebo *py.test* – v principu buď doplňují interně dostupné testovací prostředky o některé vlastnosti nebo je úplně nahrazují prostředky s jinou filozofií testování

V dalším se z rychlíku podíváme na základní použití modulu `unittest`.

> → PS: Největší nedostatek modulu `unittest` – nemožnost automatického vyhledání a spuštění všech dostupných testů v daném modulu – byla napravena po neuvěřitelně dlouhé době až v Python'u 3.2 (resp. 2.7). Používáte-li

starší verze Python'u, nejspíše skončíte u frameworku *nose* nebo podobného.

Přehled

Předpokládejme, že v modulu *something* máte nějaký kód, který potřebujete otestovat. S pomocí modulu `unittest` proto napíšete přibližně následující kód, např. do souboru *test_something.py*:

```
# import modulu zodpovědného za testy jednotek
import unittest
# import testovaného modulu
import something as sth


# třída obsahující testy
class KnownOutputs(unittest.TestCase):

    def test_1(self):
        """Popis testu 1."""
        ...

    def test_2(self):
        """Popis testu 2."""
        ...


# zajištění spuštění testů při zavolání souboru z příkazové řádky
if __name__ == '__main__':
    unittest.main()
```

Veškeré testování zajistí kód ve třídě *unittest.TestCase*, ze které vaše testovací třída dědí, vy pouze doplníte požadované testy ve formě metod, jejichž jména začínají na řetězec *test*.

Testovací metody samozřejmě musí obsahovat testovací kód. Ten je v naprosté většině případů přítomen ve formě metod *assertNĚCO*, jejichž množina se postupně s každou verzí Python'u rozrůstá. Jak z jejich názvu vyplývá, každá z těchto metod kontroluje své vstupy a jimi poskytnuté výstupy na nějakou vlastnost – např. metoda `assertIn(CO, V_ČEM)` kontroluje, zda se argument *CO* (který nejspíše získáte právě jako výstup volání nějaké části testovaného kódu) nachází v kolekci *V_ČEM* (na niž se může a nemusí vztahovat podobná poznámka), a není-li tomu tak, dá o tom na výstupu pěkně hlasitě vědět.

Samotné testování spustíte zavoláním tohoto souboru z příkazové řádky:

```
$ python3 test_something.py
```

Výstupem je přehled spuštěných testů, samozřejmě včetně chyb, ke kterým během jejich běhu došlo.

Takovýchto testovacích modulů (souborů) samozřejmě pro větší projekt napíšete více. Jelikož *test discovery* můžete použít až v těch úplně nejnovějších verzích Python'u, spuštění všech testů najednou v rámci jednoho *test suite* si tak musíte zařídit ručně sami.

Prakticky to znamená sloučit v rámci *TestSuite* všechny jednotlivé *TestCase*, které chcete spouštět pohromadě, a potom je pomocí např. *TextTestRunner* (který ve výchozím nastavení reportuje *TestResult* poskytnutý *TestCase*em na stadardní chybový výstup) zavolat.

Příklad 1

Bez dalšího motání okolo si ukažme rovnou reálný příklad (převzatý přímo z dokumentace):

```
import random
import unittest

class TestSequenceFunctions(unittest.TestCase):

    def setUp(self):
        self.seq = list(range(10))

    def test_shuffle(self):
        # make sure the shuffled sequence does not lose any elemen
        random.shuffle(self.seq)
        self.seq.sort()
        self.assertEqual(self.seq, list(range(10)))

    def test_choice(self):
        element = random.choice(self.seq)
        self.assertIn(element, self.seq)

    def test_sample(self):
        self.assertRaises(ValueError, random.sample, self.seq, 20)
        for element in random.sample(self.seq, 5):
            self.assertIn(element, self.seq)

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

→ O metodě `setUp()` se ještě zmíníme, ale nepřekvapivě slouží k nastavení atributu třídy *TestSequenceFunctions.seq* na hodnotu `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]` před každým jednotlivým testem.

Zavolání výše uvedeného testu *python test_random.py* na funkčnost modulu `random` vyprodukuje výstup..

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

..který snadno interpretujeme tak, že v neměřitelně krátkém čase proběhly tři testy a všechny dopadly bez jediné chybičky.

Metody assertNĚCO()

Metod *assertNĚCO()* je mnoho a prakticky s každou novou verzí Python'u jich přibývá. Ve velké většině jsou díky svým názvům samovysvětlující a často se vyskytují v souvisejících dvojicích. Ukažme si pár základních:

- `assertTrue, assertFalse`
- `assertEqual, assertNotEqual`
- `assertIn, assertNotIn`
- `assertRaises`
- `assertIs, assertIsNot`
- `assertIsNone, assertIsNotNone`
- …

Místo metod *assertNĚCO()* bychom mohli samozřejmě používat rovnou příkaz `assert`. To má však dvě nevýhody:

- Selhání v příkazu `assert` se projeví pouze vrácením výjimky `AssertionError`. Metody *assertNĚCO()* dokážou být podstatně „výmluvnější“.
- Příkazy `assert` se neprovádí, je-li Python spuštěn v optimalizovaném módu (tj. s přepínačem *-o* nebo *-oo*).

Metody setUp() a tearDown()

Jak jsme viděli v příkladu, metoda `setUp()` připraví kód, který je k dispozici každému testu před jeho spuštěním.

> → A to opravdu pro každý test znovu, takže pokud některé testy např. destruktivně operují nad datovými strukturami zde zavedenými, na jiné testy to nemá vliv, protože vždy dostanou jejich „čerstvou“ kopii.

Protějškem této metody je metoda `tearDown()`, která nepřekvapivě slouží naopak k „vyčištění“ (nejen po metodě *setUp()*) po doběhnutí každého testu.

Prakticky obě metody najdou často uplatnění např. v případě, že testované části kódu potřebují operovat nad objektem, jehož použití při testech je z

různých důvodů nereálné (např. produkční databáze nebo spojení na internet). Uvedené objekty pak v rámci metody *setUp()* připravíme ve „falešné“ podobě, která bude poskytovat funkcionalitu v míře dostatečné pro běh testů, a v rámci metody *tearDown()* se jich zase pečlivě zbavíme. Totéž platí i pro jejich atributy a jiné testované části kódu.

> → Uvedeným falešným objektům se stejnou funkcionalitou jako jejich skutečné protějšky se anglicky říká *mock objects*. Obecně nahrazování něčeho něčím jiným v uvedeném významu naleznete nejspíše pod termínem *monkey patching*.

Příklad 2

Ukažme si postup psaní testů a jim odpovídajícího kódu (plus některých zádrelů, které nás mohou cestou potkat) na jednoduchém příkladu. Naším úkolem je naprogramovat v modulu *something* dvě funkce s následující funkcionalitou:

- funkce *to_list()* by měla vracet svůj řetězcový argument předělaný na seznam
- funkce *to_bool()* by měla vrátit pravdivostní hodnotu svého argumentu

I. Testovaný modul something.py zatím neobsahuje doslova vůbec nic, pouze existuje, abychom ho mohli naimportovat do testů, které nepřekvapivě skončí s chybami (*error*, E), protože zatím nemají co testovat:

Příkazová řádka:

```
python3 test_something.py
```

Program *faze1/test_something.py* **: Výstup:**

```
import something as sth
import unittest



class KnownOutputs(unittest.Tes

    # Vrací sth.to_list() svůj
    def test_list_1(self):
        """Funkce something.to_
        out = sth.to_list("ahoj
        self.assertEqual(out, [

    # Vrací sth.to_bool() pro p
    def test_bool_1(self):
        """Funkce something.to_
        out = sth.to_bool('ahoj
        self.assertTrue(out)

    def test_bool_2(self):
        """Funkce something.to_
        self.assertFalse( sth.t



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

```
EEE
======================================
ERROR: test_bool_1 (__main__.Kn
Funkce something.to_bool() by m
--------------------------------
Traceback (most recent call las
  File "test_something.py", lin
    out = sth.to_bool('ahoj')
AttributeError: 'module' object

======================================
ERROR: test_bool_2 (__main__.Kn
Funkce something.to_bool() by m
--------------------------------
Traceback (most recent call las
  File "test_something.py", lin
    self.assertFalse( sth.to_bo
AttributeError: 'module' object

======================================
ERROR: test_list_1 (__main__.Kn
Funkce something.to_list() by m
--------------------------------
Traceback (most recent call las
  File "test_something.py", lin
    out = sth.to_list("ahoj")
AttributeError: 'module' object

--------------------------------
Ran 3 tests in 0.000s

FAILED (errors=3)
```

I. Připravme tedy uvedené dvě metody, které zatím ještě pořád nebudou nic dělat:

```
def to_list(xs):
    """Vrací svůj řetězcový argument předělaný na seznam."""

def to_bool(x):
    """Vrací pravdivostní hodnotu svého argumentu."""
```

Testy by nyní měly selhat (*failed*, F), protože už mají co testovat, ale daný kód zatím nic nedělá:

Příkazová řádka:

```
python3 test_something.py
```

Program *faze2/test_something.py* **: Výstup:**

```
import something as sth
import unittest



class KnownOutputs(unittest.Tes

    # Vrací sth.to_list() svůj
    def test_list_1(self):
        """Funkce something.to_
        out = sth.to_list("ahoj
        self.assertEqual(out, [

    # Vrací sth.to_bool() pro p
    def test_bool_1(self):
        """Funkce something.to_
        out = sth.to_bool('ahoj
        self.assertTrue(out)

    def test_bool_2(self):
        """Funkce something.to_
        self.assertFalse( sth.t



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

```
F.F
======================================
FAIL: test_bool_1 (__main__.Kno
Funkce something.to_bool() by m
--------------------------------
Traceback (most recent call las
  File "test_something.py", lin
    self.assertTrue(out)
AssertionError: None is not Tru

======================================
FAIL: test_list_1 (__main__.Kno
Funkce something.to_list() by m
--------------------------------
Traceback (most recent call las
  File "test_something.py", lin
    self.assertEqual(out, ['a',
AssertionError: None != ['a', '

--------------------------------
Ran 3 tests in 0.000s

FAILED (failures=2)
```

Tak moment – dva testy správně neprošly, ale co ten třetí?!? Problém je v tom, že funkce, která explicitně nic nevrací, vrací `None`, a to je v pravdivostním kontextu přeloženo na `False`, takže test projde... Naštěstí je zde ještě doplňkový test *test_bool_1*, který testuje opačnou podmínku a který nám tak řekně, že je pořád ještě něco špatně...

> → Budiž nám to ponaučením, že funkčnost testů ještě neznamená, že program dělá, co má.

I. Dobře, předpokládejme tedy, že testy už máme správně připravené, a naimplementujme uvedené dvě funkce:

```
def to_list(xs):
    """Vrací svůj řetězcový argument předělaný na seznam."""
    return list(xs)

def to_bool(x):
    """Vrací pravdivostní hodnotu svého argumentu."""
    return bool(x)
```

A otestujme je:

Příkazová řádka:

```
python3 test_something.py
```

Program *faze3/test_something.py* **:**

```
import something as sth
import unittest



class KnownOutputs(unittest.Tes

    # Vrací sth.to_list() svůj
    def test_list_1(self):
        """Funkce something.to_
        out = sth.to_list("ahoj
        self.assertEqual(out, [

    # Vrací sth.to_bool() pro p
    def test_bool_1(self):
        """Funkce something.to_
        out = sth.to_bool('ahoj
        self.assertTrue(out)

    def test_bool_2(self):
        """Funkce something.to_
        self.assertFalse( sth.t



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

Výstup:

```
...
--------------------------------
Ran 3 tests in 0.000s

OK
```

Úkol splněn, testy prošly. Všechno vypadá krásně...

I. ...než někoho napadne zavolat např. `to_list(123)`. No ovšem, funkce měla zpracovávat pouze řetězcové argumenty, ale jaksi nás nenapadlo odtestovat i jiný případ... Tak doplníme testy a zkusíme je pustit:

Příkazová řádka:

```
python3 test_something.py
```

Program *faze4/test_something.py* **:**

Výstup:

```
....
--------------------------------
Ran 4 tests in 0.000s

OK
```

```
import something as sth
import unittest



class KnownOutputs(unittest.Tes

    # Vrací sth.to_list() svůj
    def test_list_1(self):    →
        """Funkce something.to_
        out = sth.to_list("ahoj
        self.assertEqual(out, [

    def test_list_2(self):
        """Funkce something.to_
        self.assertRaises(TypeE

    # Vrací sth.to_bool() pro p
    def test_bool_1(self):
        """Funkce something.to_
        out = sth.to_bool('ahoj
        self.assertTrue(out)

    def test_bool_2(self):
        """Funkce something.to_
        self.assertFalse( sth.t



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

Zase to vypadá, že už jsme hotovi, ale to jenom proto, že volání `list(123)` opravdu „vyplivne“ výjimku *TypeError*. Ve skutečnosti naše funkce *to_list()* s radostí převede na seznam téměř cokoliv, co se na seznam převést dá.

Jelikož ale vrací očekávaný návratový typ – seznam, tak si asi nikdo ničeho nevšimne. Python jako dynamicky typovaný jazyk za nás tohle už sám neohlídá. Museli bychom přidat další testy a hlavně doplnit implementaci funkce *to_list()* o hlídání vstupních parametrů (např. pomocí vestavěné funkce `isinstance(ARGUMENT, str)`).