Automatické testování softwaru

Petr Pošík

Katedra kybernetiky, FEL ČVUT v Praze

OI, B4B33RPH: Řešení problémů a hry, 2016

Předpoklady:

- funkce
- moduly

Testujte svůj kód!

- Nebudete vědět, zda váš kód funguje, dokud jej neotestujete, tj. **dokud se ho nepokusíte použít!**

Příklad: `sum_digits()`

Specifikace: V modulu `tools.py`, vytvořte funkci `sum_digits(string)`, která vrátí součet všech číslic nalezených v řetězci `string`.

Řešení: Vytvoříme požadovaný modul s níže uvedeným obsahem:

In [1]:

```
%%writefile tools.py
def sum_digits(string):
    """Return the sum of all digits in the string"""
    sum = 0
    for ch in string:
        if ch in '012346789':
            sum += int(ch)
    return sum
```

```
Writing tools.py
```

A máme hotovo? Jak náš kód otestujeme?

Možnost 1: Zkusíme funkci použít v konzoli Pythonu

In [2]:

```
>>> from tools import sum_digits
>>> sum_digits('1, 2, 3, dee, dah, dee')
```

Out[2]:

```
6
```

- Vyzkoušeli jsme jediný testovací případ.
- Sami musíme vyhodnotit, zda je výsledek správný.
- Co když chceme test spustit znovu (např. poté, co jsme v testované funkci provedli nějakou změnu)?

Možnost 2: Testovací kód vložíme přímo do modulu

Kód, který jsme před chvilkou psali do konzole Pythonu, můžeme napsat přímo do modulu, který chceme testovat (nebo do odděleného modulu).

In [3]:

```
%%writefile tools2.py
def sum_digits(string):
    """Return the sum of all digits in the string"""
    sum = 0
    for ch in string:
        if ch in '012346789':
            sum += int(ch)
    return sum

if __name__ == "__main__":
    # All the code below is executed only when the file is run as a script.
    print(sum_digits('1, 2, 3, dee, dah, dee'))
```

```
Writing tools2.py
```

In [4]:

```
import tools2   # "Nothing" happens when we import the module (desired), ...
```

In [5]:

```
%run tools2.py     # ... but the testing code is executed when we run the module!
```

```
6
```

- Stále testujeme jediný případ.
- Stále musíme sami vyhodnotit, zda je vrácený výsledek správný.
- **Ale můžeme test snadno spouštět opakovaně, a to kolikrát chceme!**

Možnost 3: Automatická kontrola správnosti výsledku

Proč jen tisknout výsledek, když můžeme přímo otestovat, zda je správný!?

In [6]:

```
%%writefile tools3.py
def sum_digits(string):
    """Return the sum of all digits in the string"""
    sum = 0
    for ch in string:
        if ch in '012346789':
            sum += int(ch)
    return sum

if __name__ == "__main__":
    observed = sum_digits('1, 2, 3, dee, dah, dee')
    expected = 6
    if observed == expected:
        print('.')
    else:
        print('Test failed.')
        print('- Expected:', str(expected))
        print('- But got: ', str(observed))
```

```
Writing tools3.py
```

In [7]:

```
%run tools3.py
```

```
.
```

- Stále testujeme jediný případ.
- **Ale nemusíme složitě kontrolovat výsledek testu. Okamžitě vidíme, zda test prošel nebo selhal!**
- **A můžeme test snadno spouštět opakovaně, a to kolikrát chceme!**

Náš vlastní testovací modul!

Kontrola správnosti výsledku se dá extrahovat do funkce, která

- nám umožní psát testy jen s malým množstvím kódu navíc, a
- bude částí modulu, který lze použít opakovaně v mnoha projektech!

Vytvořme modul `testing` s funkcí `test_equal()`, která bude mít 3 parametry:

- pozorovanou (`observed`) a očekávanou (`expected`) hodnotu a
- volitelný název (`name`) testu.

Funkce vytiskne buď

- `"."`, když test projde v pořádku, nebo
- informativní zprávu, pokud test selže.

In [8]:

```
%%writefile testing.py
import sys

def quote(name):
    if name:
        name = "'" + name + "' "
    return name

def test_equal(observed, expected, name=''):
    """Compare the observed and expected results"""
    if observed == expected:
        print('.', end='')
    else:
        linenum = sys._getframe(1).f_lineno   # Get the caller's line number.
        print("\nTest {}at line {} FAILED:".format(quote(name), linenum))
        print("- Expected:", str(expected))
        print("- But got: ", str(observed))
```

```
Writing testing.py
```

Pomocí našeho modulu `testing`, můžeme upravit modul `tools` následovně:

In [9]:

```
%%writefile tools4.py
from testing import test_equal

def sum_digits(string):
    """Return the sum of all digits in the string"""
    sum = 0
    for ch in string:
        if ch in '012346789':
            sum += int(ch)
    return sum

if __name__ == "__main__":
    test_equal(sum_digits('1, 2, 3, dee, dah, dee'), 6, 'Test 1')
```

```
Writing tools4.py
```

In [10]:

```
%run tools4.py
```

```
.
```

- Stále testujeme pouze jediný případ.
- **Ale nemusíme složitě kontrolovat výsledek testu. Okamžitě vidíme, zda test prošel nebo selhal!**
- **A stačí nám trocha kódu, když chceme otestovat jeden případ!**
- **A můžeme test snadno spouštět opakovaně, a to kolikrát chceme!**

Další testovací případy

Máme-li další testovací případy, můžeme je přidat buď do

- sekce `if __name__=="__main__"` vyvíjeného modulu, nebo
- do odděleného testovacího skriptu.

Vytvořme oddělený testovací skript:

In [11]:

```
%%writefile test_tools.py
from testing import test_equal
from tools4 import *

def test_sum_digits():
    test_equal(sum_digits(''), 0, 'Test empty string')
    test_equal(sum_digits('0'), 0, 'Test 0')
    test_equal(sum_digits('1'), 1, 'Test 1')
    test_equal(sum_digits('2'), 2, 'Test 2')
    test_equal(sum_digits('3'), 3, 'Test 3')
    test_equal(sum_digits('4'), 4, 'Test 4')
    test_equal(sum_digits('5'), 5, 'Test 5')
    test_equal(sum_digits('6'), 6, 'Test 6')
    test_equal(sum_digits('7'), 7, 'Test 7')
    test_equal(sum_digits('8'), 8, 'Test 8')
    test_equal(sum_digits('9'), 9, 'Test 9')
    test_equal(sum_digits('1, 2, 3, dee, dah, dee'), 6, 'Non trivial test')

# Run the test suite
test_sum_digits()
```

```
Writing test_tools.py
```

In [12]:

```
%run test_tools.py
```

```
......
Test 'Test 5' at line 11 FAILED:
- Expected: 5
- But got:  0
.....
```

Ha! Máme chybu v naší funkci (nebo v našem testovacím kódu)! Dokážete chybu najít?

Testovací framework:

- Snadná tvorba obsáhlé sady testů.
- Snadné opakované spouštění testů.
- Snadná (vizuální) kontrola výsledků testů.
- Snadné přidání nových testů.

Další testovací frameworky

Náš modul `testing` není originální nápad. Python obsahuje několik oblíbených testovacích frameworků, např. standardní moduly

- `doctest` a
- `unittest`,

nebo frameworky třetích stran

- nosetest,
- pytest,
- ...

Testování kódu pomocí modulu `doctest`

- Vytvořte si zvyk uvádět příklady použití přímo v docstringu funkcí/tříd/metod (viz níže).
- Modul `doctest` vám umožní tyto příklady snadno spustit a zkontrolovat jejich výsledky!

In [13]:

```
%%writefile modulewithdoctests.py
def average(x,y):
    """Return the average of 2 numbers.

    >>> average(10,20)
    15.0
    >>> average(1.5, 2.0)
    1.75
    """
    return (x + y) / 2

if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

```
Writing modulewithdoctests.py
```

Když takový modul spustíte, testy se automaticky spustí a skutečné výsledky se porovnají s těmi očekávanými (také načtenými z docstringu):

In [14]:

```
%run modulewithdoctests.py
```

```
Trying:
    average(10,20)
Expecting:
    15.0
ok
Trying:
    average(1.5, 2.0)
Expecting:
    1.75
ok
1 items had no tests:
    __main__
1 items passed all tests:
   2 tests in __main__.average
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

Shrnutí

- Testování vašeho vlastního kódu je **extrémně důležité!**
- Testování validity řešení je **důležitá inženýrská schopnost a dovednost**, nejen při programování!
- Měli byste si osvojit alespoň jeden, ale raději více alternativních způsobů testování kódu.
- Znalost **testovacího frameworku**, ať už jednoduchého (jako je náš `testing`) nebo obsáhlého (jako je např. `unittest`), je nezanedbatelnou **výhodou**!
- Testovací frameworky typu `unittest` jsou běžné v mnoha jazycích. Jsou postaveny na stejné filozofii (xUnit). Naučíte-li se jej používat v jednom jazyce, snadno si jej osvojíte v dalších jazycích.

Notebook config

Následuje nastavení notebooku, ignorujte jej.

In [15]:

```
from notebook.services.config import ConfigManager
cm = ConfigManager()
cm.update('livereveal', {
              'theme': 'Simple',
              'transition': 'slide',
              'start_slideshow_at': 'selected',
              'width': 1268,
              'height': 768,
              'minScale': 1.0
})
```

Out[15]:

```
{'height': 768,
 'minScale': 1.0,
 'start_slideshow_at': 'selected',
 'theme': 'Simple',
 'transition': 'slide',
 'width': 1268}
```

In [16]:

```
%%HTML
<style>
.reveal #notebook-container { width: 90% !important; }
.CodeMirror { max-width: 100% !important; }
pre, code, .CodeMirror-code, .reveal pre, .reveal code {
    font-family: "Consolas", "Source Code Pro", "Courier New", Courier, monospace;
}
pre, code, .CodeMirror-code {
    font-size: inherit !important;
}
.reveal .code_cell {
    font-size: 130% !important;
    line-height: 130% !important;
}
</style>
```