
**CZECH TECHNICAL UNIVERSITY IN PRAGUE**

**Faculty of Electrical Engineering**
**Department of Cybernetics**


# A4B99RPH: Řešení problémů a hry

# Unit testing. Vývoj řízený testy.


Petr Pošík

Katedra kybernetiky
ČVUT FEL

# Úvod

# Cíle

- Ukázat, co jsou automatické testy a jaké výhody mají.
- Poskytnout metodu, jak lépe splnit zadání testíků na cvičeních.
- Zajistit, abyste se nelekli unit testů, které od nás dostanete k úloze spam filtr.
- Zkusit vás přesvědčit, že používání testů může být dobré i pro vás.

# Zpětná vazba z prvního testíku

Zadání: V modulu `vectors.py` rozšiřte třídu `MyVector`

- o metodu `__add__(self, other)` realizující sčítání 2 vektorů a
- o metodu `norm(self)` realizující výpočet velikosti vektoru.

# Zpětná vazba z prvního testíku

Zadání: V modulu `vectors.py` rozšiřte třídu `MyVector`

- o metodu `__add__(self, other)` realizující sčítání 2 vektorů a
- o metodu `norm(self)` realizující výpočet velikosti vektoru.

Čtěte dobře specifikace!

- Jak se měl jmenovat soubor, který se měl odevzdat?

# Zpětná vazba z prvního testíku

Zadání: V modulu `vectors.py` rozšiřte třídu `MyVector`

- o metodu `__add__(self, other)` realizující sčítání 2 vektorů a
- o metodu `norm(self)` realizující výpočet velikosti vektoru.

Čtěte dobře specifikace!

- Jak se měl jmenovat soubor, který se měl odevzdat?
- Jak se měla jmenovat třída definovaná v tomto souboru?

# Zpětná vazba z prvního testíku

Zadání: V modulu `vectors.py` rozšiřte třídu `MyVector`

- o metodu `__add__(self, other)` realizující sčítání 2 vektorů a
- o metodu `norm(self)` realizující výpočet velikosti vektoru.

Čtěte dobře specifikace!

- Jak se měl jmenovat soubor, který se měl odevzdat?
- Jak se měla jmenovat třída definovaná v tomto souboru?
- Jaké metody tato třída měla mít?

# Zpětná vazba z prvního testíku

Zadání: V modulu `vectors.py` rozšiřte třídu `MyVector`

- o metodu `__add__(self, other)` realizující sčítání 2 vektorů a
- o metodu `norm(self)` realizující výpočet velikosti vektoru.

Čtěte dobře specifikace!

- Jak se měl jmenovat soubor, který se měl odevzdat?
- Jak se měla jmenovat třída definovaná v tomto souboru?
- Jaké metody tato třída měla mít?
- Objekt jakého typu měla vracet metoda `__add__`?

# Zpětná vazba z prvního testíku



Zadání: V modulu `vectors.py` rozšiřte třídu `MyVector`

- o metodu `__add__(self, other)` realizující sčítání 2 vektorů a
- o metodu `norm(self)` realizující výpočet velikosti vektoru.

Čtěte dobře specifikace!

- Jak se měl jmenovat soubor, který se měl odevzdat?
- Jak se měla jmenovat třída definovaná v tomto souboru?
- Jaké metody tato třída měla mít?
- Objekt jakého typu měla vracet metoda `__add__`?

Proč jste tyto chyby neodhalili?

# Zpětná vazba z prvního testíku



Zadání: V modulu `vectors.py` rozšiřte třídu `MyVector`

- o metodu `__add__(self, other)` realizující sčítání 2 vektorů a
- o metodu `norm(self)` realizující výpočet velikosti vektoru.

Čtěte dobře specifikace!

- Jak se měl jmenovat soubor, který se měl odevzdat?
- Jak se měla jmenovat třída definovaná v tomto souboru?
- Jaké metody tato třída měla mít?
- Objekt jakého typu měla vracet metoda `__add__`?

Proč jste tyto chyby neodhalili?

Jak otestovat vlastní kód?

# Ukázka na třídě MyVector



Demo

# Python Shell



Spustit Python shell a zkusit svůj kód použít:

```
>>> from vectors import MyVector
>>> a = MyVector([1,1,1])
>>> b = MyVector([1,2,3])
>>> c = a+b
>>> type(c)
<class 'vectors.MyVector'>
>>> c.get_vector()
[2, 3, 4]
```

# Python Shell



Spustit Python shell a zkusit svůj kód použít:

```
>>> from vectors import MyVector
>>> a = MyVector([1,1,1])
>>> b = MyVector([1,2,3])
>>> c = a+b
>>> type(c)
<class 'vectors.MyVector'>
>>> c.get_vector()
[2, 3, 4]
```

Snadno odhalíte všechny výše zmíněné chyby, jenže:

- Při každé změně v kódu dá hodně práce test znovu spustit.
- Správnost výsledků musíte ohodnotit vy sami.
- Navíc:
  - Někdy je nutné přepnout se do pracovního adresáře (`import os; os.chdir()`).
  - Bývají problémy s re-importem již jednou importovaného modulu.
    - V Pythonu 2x: `reload(module)`
    - V Pythonu 3x: `import imp; imp.reload(module)`
    - Ani v jednom případě nefunguje re-import zcela spolehlivě.

# Manuální testy

Využít sekci `if __name__ == '__main__'`: ke spuštění testu:

```
class MyVector:
    ...

if __name__ == "__main__":
    from vectors import MyVector
    a = MyVector([1,1,1])
    b = MyVector([1,2,3])
    c = a+b
    print(type(c))
    print(c.get_vector())
```

# Manuální testy



Využít sekci `if __name__ == '__main__':` ke spuštění testu:

```
class MyVector:
    ...

if __name__ == "__main__":
    from vectors import MyVector
    a = MyVector([1,1,1])
    b = MyVector([1,2,3])
    c = a+b
    print(type(c))
    print(c.get_vector())
```

Rychlé opakované spuštění testu je velmi snadné! Ale:

- Správnost výsledků musíte stále hodnotit sami.
- Je-li třeba otestovat několik rysů funkce/třídy:
  - Nelze je testovat po jednom (leda zakomentováním ostatních).
  - Je obtížné udržet v testech pořádek.

# Automatické testování

Využít metod automatického testování:

- doctest, unittest, či jiný framework
- lze spouštět i velké množství testů s automatickou kontrolou výsledků

# Binární matice záměn

# Z úlohy “Spam filter”



Předpokládejme:

- máme sadu emailů uložených v souborech
- pro každý email z této sady víme, zda je to spam nebo ham
- máme jakýkoli funkční spam filter
- pro každý email z naší sady víme, zda jej filtr klasifikuje jako spam nebo ham

# Z úlohy “Spam filter”



Předpokládejme:

- máme sadu emailů uložených v souborech
- pro každý email z této sady víme, zda je to spam nebo ham
- máme jakýkoli funkční spam filter
- pro každý email z naší sady víme, zda jej filtr klasifikuje jako spam nebo ham

Jakých chyb se může spam filtr dopustit?

# Z úlohy “Spam filter”



Předpokládejme:

- máme sadu emailů uložených v souborech
- pro každý email z této sady víme, zda je to spam nebo ham
- máme jakýkoli funkční spam filter
- pro každý email z naší sady víme, zda jej filtr klasifikuje jako spam nebo ham

Jakých chyb se může spam filtr dopustit? *Matice záměn!*

| | V datové sadě pozitivních | negativních |
|---|---|---|
| Pozitivní předpověď | $TP$ | $FP$ |
| Negativní předpověď | $FN$ | $TN$ |

# Z úlohy “Spam filter”



Předpokládejme:

- máme sadu emailů uložených v souborech
- pro každý email z této sady víme, zda je to spam nebo ham
- máme jakýkoli funkční spam filter
- pro každý email z naší sady víme, zda jej filtr klasifikuje jako spam nebo ham

Jakých chyb se může spam filtr dopustit? *Matice záměn!*

| | V datové sadě pozitivních | V datové sadě negativních |
|---|---|---|
| Pozitivní předpověď | $TP$ | $FP$ |
| Negativní předpověď | $FN$ | $TN$ |

**True positives ($TP$)**: počet případů klasifikátorem *správně* označených jako *pozitivní*.

**False positives ($FP$)**: počet případů klasifikátorem *chybně* označených jako *pozitivní*.

**False negatives ($FN$)**: počet případů klasifikátorem *chybně* označených jako *negativní*.

**True negatives ($TN$)**: počet případů klasifikátorem *správně* označených jako *negativní*.

# Z úlohy “Spam filter”



Předpokládejme:

- máme sadu emailů uložených v souborech
- pro každý email z této sady víme, zda je to spam nebo ham
- máme jakýkoli funkční spam filter
- pro každý email z naší sady víme, zda jej filtr klasifikuje jako spam nebo ham

Jakých chyb se může spam filtr dopustit? *Matice záměn!*

| | V datové sadě pozitivních | V datové sadě negativních |
|---|---|---|
| Pozitivní předpověď | $TP$ | $FP$ |
| Negativní předpověď | $FN$ | $TN$ |

**True positives ($TP$)**: počet případů klasifikátorem *správně* označených jako *pozitivní*.
**False positives ($FP$)**: počet případů klasifikátorem *chybně* označených jako *pozitivní*.
**False negatives ($FN$)**: počet případů klasifikátorem *chybně* označených jako *negativní*.
**True negatives ($TN$)**: počet případů klasifikátorem *správně* označených jako *negativní*.

**Míra kvality** je pak nějakou funkcí $TP$, $TN$, $FP$ a $FN$.

# Binární matice záměn



## Binary confusion matrix, BCM:

- Takový "lepší" čítač (čtveřice čítačů).
- Cíl: Ze slovníků `truth` a `prediction` napočítat matici záměn.

```
>>> truth = {
        'email1': 'OK',
        'email2': 'OK',
        'email3': 'SPAM',
        ...
        }
```

```
>>> prediction = {
        'email1': 'SPAM',
        'email2': 'OK',
        'email3': 'SPAM',
        ...
        }
```

Požadavky na BCF:

# Binární matice záměn



## Binary confusion matrix, BCM:

- Takový "lepší" čítač (čtveřice čítačů).
- Cíl: Ze slovníků `truth` a `prediction` napočítat matici záměn.

```
>>> truth = {
        'email1': 'OK',
        'email2': 'OK',
        'email3': 'SPAM',
        ...
        }
```

```
>>> prediction = {
        'email1': 'SPAM',
        'email2': 'OK',
        'email3': 'SPAM',
        ...
        }
```

Požadavky na BCF:

- Lze nastavit libovolný kód pro spam a ham (zde např OK a SPAM).
- Metoda `as_dict()` vrátí čítače ve formě slovníku.
- Po vytvoření objektu jsou všechny čítače vynulované.
- Zavolám-li metodu `update('SPAM', 'SPAM')`, inkrementuje se čítač TP *a hodnota ostatních se nezmění*.
- ...
- Zavolám-li metodu `update()` s nesprávným argumentem, vyhodí se výjimka `ValueError`.
- Zavolám-li metodu `compute_from_dicts(truth, prediction)`, čítače TP, FP, TN, FN se správně aktualizují.

# Ukázka vývoje BCF s testy



Demo

# Testy s modulem `unittest`

```
import unittest
from confmat import BinaryConfusionMatrix
SPAM_TAG = 'SPAM'
HAM_TAG = 'HAM'

class BinaryConfusionMatrixTest(unittest.TestCase):

    def setUp(self):
        self.cm = BinaryConfusionMatrix(
            pos_tag=SPAM_TAG, neg_tag=HAM_TAG)

    def test_countersAreZero_afterCreation(self):
        # Exercise the SUT
        cmdict = self.cm.as_dict()
        # Assert
        self.assertDictEqual(cmdict,
            {'tp': 0, 'tn': 0, 'fp': 0, 'fn': 0})

    def test_updatesTPcorrectly(self):
        # Exercise the SUT
        self.cm.update(SPAM_TAG, SPAM_TAG)
        # Assert
        self.assertDictEqual(self.cm.as_dict(),
            {'tp': 1, 'tn': 0, 'fp': 0, 'fn': 0})

    def test_updatesTNcorrectly(self):
        # Exercise the SUT
        self.cm.update(HAM_TAG, HAM_TAG)
        # Assert
        self.assertDictEqual(self.cm.as_dict(),
            {'tp': 0, 'tn': 1, 'fp': 0, 'fn': 0})

    def test_updatesFPcorrectly(self):
        # Exercise the SUT
        self.cm.update(HAM_TAG, SPAM_TAG)
        # Assert
        self.assertDictEqual(self.cm.as_dict(),
            {'tp': 0, 'tn': 0, 'fp': 1, 'fn': 0})

    def test_updatesFNcorrectly(self):
        # Exercise the SUT
        self.cm.update(SPAM_TAG, HAM_TAG)
        # Assert
        self.assertDictEqual(self.cm.as_dict(),
            {'tp': 0, 'tn': 0, 'fp': 0, 'fn': 1})

    def test_update_raisesValueError_forWrongTruthValue(self):
        # Assert and exercise the SUT
        with self.assertRaises(ValueError):
            self.cm.update('a bad value', SPAM_TAG)

    def test_update_raisesValueError_forWrongPredictionValue(self
        # Assert and exercise the SUT
        with self.assertRaises(ValueError):
            self.cm.update(SPAM_TAG, 'a bad value')

    def test_computeFromDicts_allCasesOnce(self):
        # Prepare fixture
        truth = {1: SPAM_TAG, 2: SPAM_TAG, 3: HAM_TAG, 4: HAM_TAG
        prediction = {1: SPAM_TAG, 2: HAM_TAG, 3: SPAM_TAG, 4: HA
        # Excercise the SUT
        self.cm.compute_from_dicts(truth, prediction)
        # Assert
        self.assertDictEqual(self.cm.as_dict(),
            {'tp': 1, 'tn': 1, 'fp': 1, 'fn': 1})

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

# Automatizované testování

**Zpracováno podle**
**Gerard Meszarosz: *xUnit Test Patterns: Refactoring Test Code*,**
**Addison-Wesley, 2007.**

# Testování

*Kvalita* softwaru z pohledu testování:

- Jak dobře kód splňuje specifikace?

# Testování



*Kvalita* softwaru z pohledu testování:

- Jak dobře kód splňuje specifikace?

Testování z pohledu QA týmu (acceptance tests, functional tests):

- Testujeme, protože jsme si jistí, že kód obsahuje chyby! (Nesplňuje specifikace zákazníka.)
- Testujeme poté, co je kód hotový.
- Obvykle black-box testování.
- Testování je spíš *měření* kvality softwaru, nikoli způsob, jak napsat kvalitní software.
- Zpětná vazba přichází příliš pozdě.
- V minulosti prováděny převážně ručně.

# Testování



*Kvalita* softwaru z pohledu testování:

- Jak dobře kód splňuje specifikace?

Testování z pohledu QA týmu (acceptance tests, functional tests):

- Testujeme, protože jsme si jistí, že kód obsahuje chyby! (Nesplňuje specifikace zákazníka.)
- Testujeme poté, co je kód hotový.
- Obvykle black-box testování.
- Testování je spíš *měření* kvality softwaru, nikoli způsob, jak napsat kvalitní software.
- Zpětná vazba přichází příliš pozdě.
- V minulosti prováděny převážně ručně.

Testování z pohledu programátora (unit tests, integration tests):

- Testuji, protože si chci být jistý, že jednotka, na které právě pracuji, dělá to, co po ní chci. (Splňuje požadavky, které vznikly v důsledku designu architektury softwaru.)
- Obvykle white-box testování.
- V minulosti většinou dočasný kód, který se po otestování zahodil.

# Automatizované testy: F.I.R.S.T.

Automatizované testy by měly být F.I.R.S.T.

# Automatizované testy: F.I.R.S.T.

Automatizované testy by měly být F.I.R.S.T.

## Fast

- Pomalé testy $\longrightarrow$ nebudete je spouštět často $\longrightarrow$ chyby odhalíte pozdě

# Automatizované testy: F.I.R.S.T.



Automatizované testy by měly být F.I.R.S.T.

## Fast

- Pomalé testy $\longrightarrow$ nebudete je spouštět často $\longrightarrow$ chyby odhalíte pozdě

## Independent

- Jeden test by neměl nastavovat podmínky pro další test.
- Musí jít spustit každý test samostatně a celou sadu testů v jakémkoli pořadí.
- Závislé testy $\longrightarrow$ jedna chyba spustí celý řetězec chyb v navazujících testech $\longrightarrow$ složité hledání chyby.

# Automatizované testy: F.I.R.S.T.



Automatizované testy by měly být F.I.R.S.T.

## Fast

- Pomalé testy ⟶ nebudete je spouštět často ⟶ chyby odhalíte pozdě

## Independent

- Jeden test by neměl nastavovat podmínky pro další test.
- Musí jít spustit každý test samostatně a celou sadu testů v jakémkoli pořadí.
- Závislé testy ⟶ jedna chyba spustí celý řetězec chyb v navazujících testech ⟶ složité hledání chyby.

## Repeatable

- Možnost *zopakovat* testy kýmkoli a kdekoli se stejným výsledkem.
- Testy lze spustit jen někde ⟶ budou se pouštět zřídka ⟶ chyby odhalíte pozdě

# Automatizované testy: F.I.R.S.T.



Automatizované testy by měly být F.I.R.S.T.

## Fast

- Pomalé testy ⟶ nebudete je spouštět často ⟶ chyby odhalíte pozdě

## Independent

- Jeden test by neměl nastavovat podmínky pro další test.
- Musí jít spustit každý test samostatně a celou sadu testů v jakémkoli pořadí.
- Závislé testy ⟶ jedna chyba spustí celý řetězec chyb v navazujících testech ⟶ složité hledání chyby.

## Repeatable

- Možnost *zopakovat* testy kýmkoli a kdekoli se stejným výsledkem.
- Testy lze spustit jen někde ⟶ budou se pouštět zřídka ⟶ chyby odhalíte pozdě

## Self-validating

- Dvoustavový výstup ⟶ snadné ověřit, zda test prošel nebo selhal.
- Složitý (dlouhý) výstup, který je nutno "ručně" zkontrolovat ⟶ málo časté testování ⟶ pozdní odhalení chyb.

# Automatizované testy: F.I.R.S.T.



Automatizované testy by měly být F.I.R.S.T.

## Fast

- Pomalé testy ⟶ nebudete je spouštět často ⟶ chyby odhalíte pozdě

## Independent

- Jeden test by neměl nastavovat podmínky pro další test.
- Musí jít spustit každý test samostatně a celou sadu testů v jakémkoli pořadí.
- Závislé testy ⟶ jedna chyba spustí celý řetězec chyb v navazujících testech ⟶ složité hledání chyby.

## Repeatable

- Možnost *zopakovat* testy kýmkoli a kdekoli se stejným výsledkem.
- Testy lze spustit jen někde ⟶ budou se pouštět zřídka ⟶ chyby odhalíte pozdě

## Self-validating

- Dvoustavový výstup ⟶ snadné ověřit, zda test prošel nebo selhal.
- Složitý (dlouhý) výstup, který je nutno "ručně" zkontrolovat ⟶ málo časté testování ⟶ pozdní odhalení chyb.

## Timely

- Testy by měly být psány včas, ideálně před produkčním kódem.
- Testy psané po produkčním kódu ⟶ kód se špatně testuje ⟶ nebudete se chtít s jeho testováním zdržovat.

# Modul doctest



- specialita Pythonu (opravte mě, pokud se pletu)
- velmi vhodný pro jednoduché testy nevyžadující žádnou přípravu a úklid
- složitější testy lze dělat také, ale někomu to přijde . . . nepřirozené

```
class PrimesGenerator:
    """Prime numbers generator.

    >>> pg = PrimesGenerator()
    >>> pg.get_primes_up_to(1)
    []
    >>> pg.get_primes_up_to(2)
    [2]
    >>> pg.get_primes_up_to(3)
    [2, 3]
    >>> pg.get_primes_up_to(4)
    [2, 3]
    >>> pg.get_primes_up_to(5)
    [2, 3, 5]
    >>> pg.get_primes_up_to(20)
    [2, 3, 5, 7, 11, 13, 17, 19]
    """
    ...

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

# xUnit Framework

- Standardní testovací framework
- Implementován v mnoha jazycích (naučte se ho, bude se vám hodit)
- V Pythonu implementován jako modul `unittest`.

# xUnit Framework

- Standardní testovací framework
- Implementován v mnoha jazycích (naučte se ho, bude se vám hodit)
- V Pythonu implementován jako modul `unittest`.



```
import unittest
from primes3 import PrimesGenerator

class PrimesGeneratorTest(unittest.TestCase):

    known_values = ( ( 0, [] ),
                     ( 1, [] ),
                     ( 2, [2] ),
                     ( 3, [2,3] ),
                     ( 4, [2,3] ),
                     ( 5, [2,3,5] ),
                     ( 7, [2,3,5,7]),
                     ( 20, [2,3,5,7,11,13,17,19] ))

    def setUp(self):
        self.pg = PrimesGenerator()

    def test_get_primes_up_to(self):
        for limit, expected in self.known_values:
            observed = self.pg.get_primes_up_to(limit)
            self.assertEqual(observed, expected)
    ...

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

# Vývoj řízený testy

# TDD: Vývoj řízený testy

Tři zákony TDD (Test-driven development):

1. Nenapíšeš ani kousek produkčního kódu, aniž bys předtím napsal selhávající unit test.

# TDD: Vývoj řízený testy

Tři zákony TDD (Test-driven development):

1. Nenapíšeš ani kousek produkčního kódu, aniž bys předtím napsal selhávající unit test.
2. Nenapíšeš větší část unit testu, než je potřebná k selhání (chybě).

# TDD: Vývoj řízený testy

Tři zákony TDD (Test-driven development):

1. Nenapíšeš ani kousek produkčního kódu, aniž bys předtím napsal selhávající unit test.
2. Nenapíšeš větší část unit testu, než je potřebná k selhání (chybě).
3. Nenapíšeš větší část produkčního kódu, než je potřebná ke splnění aktuálně selhávajícího unit testu.

# TDD: Vývoj řízený testy



Tři zákony TDD (Test-driven development):

1. Nenapíšeš ani kousek produkčního kódu, aniž bys předtím napsal selhávající unit test.
2. Nenapíšeš větší část unit testu, než je potřebná k selhání (chybě).
3. Nenapíšeš větší část produkčního kódu, než je potřebná ke splnění aktuálně selhávajícího unit testu.

Výsledek těchto pravidel:

- velmi krátký cyklus, v němž střídavě hrajete
  - roli zákazníka, který říká, co se má dělat (píšete test), a
  - roli programátora, který říká, jak se to má dělat (upravujete kód).
- Testy a produkční kód se píší *společně* (testy o pár sekund napřed).
- Testy pak pokrývají všechen produkční kód!

# TDD Ukázka

Vytvořte funkci/metodu třídy na faktorizaci čísla na prvočíselné činitele.

- Vstup: číslo, které chceme rozložit
- Výstup: seznam prvočísel, jejichž součin je roven vstupnímu číslu

# TDD Ukázka

Vytvořte funkci/metodu třídy na faktorizaci čísla na prvočíselné činitele.

- Vstup: číslo, které chceme rozložit
- Výstup: seznam prvočísel, jejichž součin je roven vstupnímu číslu

Jak byste postupovali?

# TDD Ukázka: Úvodní fáze

Zakládáme `test_factorize.py`

```
import unittest
from factorization import factorize
```

# TDD Ukázka: Úvodní fáze

## Zakládáme `test_factorize.py`

```
import unittest
from factorization import factorize
```

## Po spuštění `test_factorize.py`:

```
Traceback (most recent call last):
  File "<string>", line 2, in <fragment>
builtins.ImportError: No module named factorization
```

# TDD Ukázka: Úvodní fáze

Zakládáme `test_factorize.py`

```
import unittest
from factorization import factorize
```

Po spuštění `test_factorize.py`:

```
Traceback (most recent call last):
  File "<string>", line 2, in <fragment>
builtins.ImportError: No module named factorization
```

---

Zakládáme prázdný `factorization.py`

# TDD Ukázka: Úvodní fáze

## Zakládáme `test_factorize.py`

```
import unittest
from factorization import factorize
```

## Po spuštění `test_factorize.py`:

```
Traceback (most recent call last):
  File "<string>", line 2, in <fragment>
builtins.ImportError: No module named factorization
```

---

## Zakládáme prázdný `factorization.py`

## Po spuštění `test_factorize.py`:

```
Traceback (most recent call last):
  File "<string>", line 2, in <fragment>
builtins.ImportError: cannot import name factorize
```

# TDD Ukázka: Úvodní fáze

Zakládáme `test_factorize.py`

```
import unittest
from factorization import factorize
```

Po spuštění `test_factorize.py`:

```
Traceback (most recent call last):
  File "<string>", line 2, in <fragment>
builtins.ImportError: No module named factorization
```

---

Zakládáme prázdný `factorization.py`

Po spuštění `test_factorize.py`:

```
Traceback (most recent call last):
  File "<string>", line 2, in <fragment>
builtins.ImportError: cannot import name factorize
```

---

Upravujeme `factorization.py`:

```
def factorize():
  pass
```

# TDD Ukázka: Úvodní fáze

## Zakládáme `test_factorize.py`

```
import unittest
from factorization import factorize
```

## Po spuštění `test_factorize.py`:

```
Traceback (most recent call last):
  File "<string>", line 2, in <fragment>
builtins.ImportError: No module named factorization
```

---

## Zakládáme prázdný `factorization.py`

## Po spuštění `test_factorize.py`:

```
Traceback (most recent call last):
  File "<string>", line 2, in <fragment>
builtins.ImportError: cannot import name factorize
```

---

## Upravujeme `factorization.py`:

```
def factorize():
  pass
```

## Po spuštění `test_factorize.py`:

```
--- Zadny vystup, kod bez chyby. ---
```

# TDD Ukázka: Úvodní fáze

## Zakládáme `test_factorize.py`

```
import unittest
from factorization import factorize
```

## Po spuštění `test_factorize.py`:

```
Traceback (most recent call last):
  File "<string>", line 2, in <fragment>
builtins.ImportError: No module named factorization
```

---

## Zakládáme prázdný `factorization.py`

## Po spuštění `test_factorize.py`:

```
Traceback (most recent call last):
  File "<string>", line 2, in <fragment>
builtins.ImportError: cannot import name factorize
```

---

## Upravujeme `factorization.py`:

```
def factorize():
  pass
```

## Po spuštění `test_factorize.py`:

```
--- Zadny vystup, kod bez chyby. ---
```

---

## Upravujeme `test_factorize.py`

```
import unittest
from factorization import factorize

class FactorizeTest(unittest.TestCase):
  pass

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

# TDD Ukázka: Úvodní fáze

## Zakládáme `test_factorize.py`

```
import unittest
from factorization import factorize
```

## Po spuštění `test_factorize.py`:

```
Traceback (most recent call last):
  File "<string>", line 2, in <fragment>
builtins.ImportError: No module named factorization
```

---

## Zakládáme prázdný `factorization.py`

## Po spuštění `test_factorize.py`:

```
Traceback (most recent call last):
  File "<string>", line 2, in <fragment>
builtins.ImportError: cannot import name factorize
```

---

## Upravujeme `factorization.py`:

```
def factorize():
  pass
```

## Po spuštění `test_factorize.py`:

```
--- Zadny vystup, kod bez chyby. ---
```

---

## Upravujeme `test_factorize.py`

```
import unittest
from factorization import factorize

class FactorizeTest(unittest.TestCase):
  pass

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

## Po spuštění `test_factorize.py`:

```
----------------------------------------------------------------------
Ran 0 tests in 0.000s

OK
builtins.SystemExit: False
```

# TDD Ukázka: Test faktorizace čísla 2

Upravujeme `test_factorize.py`

```
class FactorizeTest(unittest.TestCase):

  def test_two(self):
    observed = factorize(2)
    self.assertEqual(observed, [2])
```

# TDD Ukázka: Test faktorizace čísla 2

Upravujeme `test_factorize.py`

```
class FactorizeTest(unittest.TestCase):

  def test_two(self):
    observed = factorize(2)
    self.assertEqual(observed, [2])
```

Po spuštění `test_factorize.py`:

```
E
======================================================================
ERROR: test_one (__main__.FactorizeTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<wingdb_compile>", line 7, in test_one
TypeError: factorize() takes no arguments (1 given)
----------------------------------------------------------------------
Ran 1 test in 0.000s
```

---

Upravujeme `factorization.py`:

```
def factorize(multiple):
  pass
```

# TDD Ukázka: Test faktorizace čísla 2

## Upravujeme `test_factorize.py`

```
class FactorizeTest(unittest.TestCase):

  def test_two(self):
    observed = factorize(2)
    self.assertEqual(observed, [2])
```

## Po spuštění `test_factorize.py`:

```
E
======================================================================
ERROR: test_one (__main__.FactorizeTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<wingdb_compile>", line 7, in test_one
TypeError: factorize() takes no arguments (1 given)
----------------------------------------------------------------------
Ran 1 test in 0.000s
```

---

## Upravujeme `factorization.py`:

```
def factorize(multiple):
  pass
```

```
F
======================================================================
FAIL: test_one (__main__.FactorizeTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<wingdb_compile>", line 8, in test_one
AssertionError: None != [2]
----------------------------------------------------------------------
Ran 1 test in 0.000s
```

# TDD Ukázka: Test faktorizace čísla 2

## Upravujeme `test_factorize.py`

```
class FactorizeTest(unittest.TestCase):

  def test_two(self):
    observed = factorize(2)
    self.assertEqual(observed, [2])
```

## Po spuštění `test_factorize.py`:

```
E
======================================================================
ERROR: test_one (__main__.FactorizeTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<wingdb_compile>", line 7, in test_one
TypeError: factorize() takes no arguments (1 given)
----------------------------------------------------------------------
Ran 1 test in 0.000s
```

---

## Upravujeme `factorization.py`:

```
def factorize(multiple):
  pass
```

```
F
======================================================================
FAIL: test_one (__main__.FactorizeTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<wingdb_compile>", line 8, in test_one
AssertionError: None != [2]
----------------------------------------------------------------------
Ran 1 test in 0.000s
```

---

## Upravujeme `factorization.py`:

```
def factorize(multiple):
  return [2]
```

# TDD Ukázka: Test faktorizace čísla 2

## Upravujeme `test_factorize.py`

```
class FactorizeTest(unittest.TestCase):

  def test_two(self):
    observed = factorize(2)
    self.assertEqual(observed, [2])
```

## Po spuštění `test_factorize.py`:

```
E
======================================================================
ERROR: test_one (__main__.FactorizeTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<wingdb_compile>", line 7, in test_one
TypeError: factorize() takes no arguments (1 given)
----------------------------------------------------------------------
Ran 1 test in 0.000s
```

---

## Upravujeme `factorization.py`:

```
def factorize(multiple):
  pass
```

```
F
======================================================================
FAIL: test_one (__main__.FactorizeTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<wingdb_compile>", line 8, in test_one
AssertionError: None != [2]
----------------------------------------------------------------------
Ran 1 test in 0.000s
```

---

## Upravujeme `factorization.py`:

```
def factorize(multiple):
  return [2]
```

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s
```

# TDD Ukázka: Test faktorizace čísla 3

Upravujeme `test_factorize.py`

```
  def test_three(self):
    observed = factorize(3)
    self.assertEqual(observed, [3])
```

# TDD Ukázka: Test faktorizace čísla 3

Upravujeme `test_factorize.py`

```
  def test_three(self):
    observed = factorize(3)
    self.assertEqual(observed, [3])
```

Po spuštění `test_factorize.py`:

```
F.
======================================================================
FAIL: test_three (__main__.FactorizeTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<wingdb_compile>", line 12, in test_three
AssertionError: Lists differ: [2] != [3]

First differing element 0:
2
3

- [2]
+ [3]

----------------------------------------------------------------------
Ran 2 tests in 0.016s
```

---

Upravujeme `factorization.py`:

```
def factorize(multiple):
  return [multiple]
```

# TDD Ukázka: Test faktorizace čísla 3

## Upravujeme `test_factorize.py`

```
  def test_three(self):
    observed = factorize(3)
    self.assertEqual(observed, [3])
```

## Po spuštění `test_factorize.py`:

```
F.
======================================================================
FAIL: test_three (__main__.FactorizeTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<wingdb_compile>", line 12, in test_three
AssertionError: Lists differ: [2] != [3]

First differing element 0:
2
3

- [2]
+ [3]

----------------------------------------------------------------------
Ran 2 tests in 0.016s
```

---

## Upravujeme `factorization.py`:

```
def factorize(multiple):
  return [multiple]
```

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s
```

# TDD Ukázka: Test faktorizace čísla 4

Upravujeme `test_factorize.py`

```
  def test_four(self):
    observed = factorize(4)
    self.assertEqual(observed, [2,2])
```

# TDD Ukázka: Test faktorizace čísla 4

## Upravujeme `test_factorize.py`

```
  def test_four(self):
    observed = factorize(4)
    self.assertEqual(observed, [2,2])
```

## Po spuštění `test_factorize.py`:

```
F..
======================================================================
FAIL: test_four (__main__.FactorizeTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<wingdb_compile>", line 16, in test_four
AssertionError: Lists differ: [4] != [2, 2]
[...snip...]
----------------------------------------------------------------------
Ran 3 tests in 0.000s
```

## Upravujeme `factorization.py`:

```
def factorize(multiple):
  factors = []
  while multiple % 2 == 0:
    factors.append(2)
    multiple /= 2
  return factors
```

# TDD Ukázka: Test faktorizace čísla 4

## Upravujeme `test_factorize.py`

```
  def test_four(self):
    observed = factorize(4)
    self.assertEqual(observed, [2,2])
```

## Po spuštění `test_factorize.py`:

```
F..
======================================================================
FAIL: test_four (__main__.FactorizeTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<wingdb_compile>", line 16, in test_four
AssertionError: Lists differ: [4] != [2, 2]
[...snip...]
----------------------------------------------------------------------
Ran 3 tests in 0.000s
```

## Upravujeme `factorization.py`:

```
def factorize(multiple):
  factors = []
  while multiple % 2 == 0:
    factors.append(2)
    multiple /= 2
  return factors
```

```
.F.
======================================================================
FAIL: test_three (__main__.FactorizeTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<wingdb_compile>", line 12, in test_three
AssertionError: Lists differ: [] != [3]
[...snip...]
----------------------------------------------------------------------
Ran 3 tests in 0.016s
```

# TDD Ukázka: Test faktorizace čísla 4

## Upravujeme `test_factorize.py`

```
  def test_four(self):
    observed = factorize(4)
    self.assertEqual(observed, [2,2])
```

## Po spuštění `test_factorize.py`:

```
F..
======================================================================
FAIL: test_four (__main__.FactorizeTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<wingdb_compile>", line 16, in test_four
AssertionError: Lists differ: [4] != [2, 2]
[...snip...]
----------------------------------------------------------------------
Ran 3 tests in 0.000s
```

## Upravujeme `factorization.py`:

```
def factorize(multiple):
  factors = []
  while multiple % 2 == 0:
    factors.append(2)
    multiple /= 2
  return factors
```

```
.F.
======================================================================
FAIL: test_three (__main__.FactorizeTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<wingdb_compile>", line 12, in test_three
AssertionError: Lists differ: [] != [3]
[...snip...]
----------------------------------------------------------------------
Ran 3 tests in 0.016s
```

## Upravujeme `factorization.py`:

```
def factorize(multiple):
  factors = []
  while multiple % 2 == 0:
    factors.append(2)
    multiple /= 2
  if multiple != 1:
    factors.append(multiple)
  return factors
```

# TDD Ukázka: Test faktorizace čísla 4

## Upravujeme `test_factorize.py`

```
  def test_four(self):
    observed = factorize(4)
    self.assertEqual(observed, [2,2])
```

## Po spuštění `test_factorize.py`:

```
F..
======================================================================
FAIL: test_four (__main__.FactorizeTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<wingdb_compile>", line 16, in test_four
AssertionError: Lists differ: [4] != [2, 2]
[...snip...]
----------------------------------------------------------------------
Ran 3 tests in 0.000s
```

## Upravujeme `factorization.py`:

```
def factorize(multiple):
  factors = []
  while multiple % 2 == 0:
    factors.append(2)
    multiple /= 2
  return factors
```

```
.F.
======================================================================
FAIL: test_three (__main__.FactorizeTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<wingdb_compile>", line 12, in test_three
AssertionError: Lists differ: [] != [3]
[...snip...]
----------------------------------------------------------------------
Ran 3 tests in 0.016s
```

## Upravujeme `factorization.py`:

```
def factorize(multiple):
  factors = []
  while multiple % 2 == 0:
    factors.append(2)
    multiple /= 2
  if multiple != 1:
    factors.append(multiple)
  return factors
```

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

# TDD Ukázka: Test faktorizace čísla 5

Upravujeme `test_factorize.py`

```
  def test_five(self):
    observed = factorize(5)
    self.assertEqual(observed, [5])
```

# TDD Ukázka: Test faktorizace čísla 5

Upravujeme `test_factorize.py`

```
  def test_five(self):
    observed = factorize(5)
    self.assertEqual(observed, [5])
```

Po spuštění `test_factorize.py`:

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

# TDD Ukázka: Test faktorizace čísla 6

Upravujeme `test_factorize.py`

```
def test_six(self):
  observed = factorize(6)
  self.assertEqual(observed, [2,3])
```

# TDD Ukázka: Test faktorizace čísla 6

Upravujeme `test_factorize.py`

```
  def test_six(self):
    observed = factorize(6)
    self.assertEqual(observed, [2,3])
```

Po spuštění `test_factorize.py`:

```
.....
----------------------------------------------------------------------
Ran 5 tests in 0.000s
```

# TDD Ukázka: Test faktorizace čísla 6

Upravujeme `test_factorize.py`

```
  def test_six(self):
    observed = factorize(6)
    self.assertEqual(observed, [2,3])
```

Po spuštění `test_factorize.py`:

```
.....
----------------------------------------------------------------------
Ran 5 tests in 0.000s
```

Test faktorizace čísla 7 vynecháváme, je to stejný případ, jako pro 3 a 5.

# TDD Ukázka: Test faktorizace čísla 8

Upravujeme `test_factorize.py`

```
  def test_eight(self):
    observed = factorize(8)
    self.assertEqual(observed, [2,2,2])
```

# TDD Ukázka: Test faktorizace čísla 8

Upravujeme `test_factorize.py`

```
  def test_eight(self):
    observed = factorize(8)
    self.assertEqual(observed, [2,2,2])
```

Po spuštění `test_factorize.py`:

```
......
----------------------------------------------------------------------
Ran 6 tests in 0.000s
```

# TDD Ukázka: Test faktorizace čísla 9

Upravujeme `test_factorize.py`

```
  def test_nine(self):
    observed = factorize(9)
    self.assertEqual(observed, [3,3])
```

# TDD Ukázka: Test faktorizace čísla 9

## Upravujeme `test_factorize.py`

```
  def test_nine(self):
    observed = factorize(9)
    self.assertEqual(observed, [3,3])
```

## Po spuštění `test_factorize.py`:

```
...F...
======================================================================
FAIL: test_nine (__main__.FactorizeTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<wingdb_compile>", line 32, in test_nine
AssertionError: Lists differ: [9] != [3, 3]
[...snip...]
----------------------------------------------------------------------
Ran 7 tests in 0.000s
```

# TDD Ukázka: Test faktorizace čísla 9

## Upravujeme `test_factorize.py`

```
  def test_nine(self):
    observed = factorize(9)
    self.assertEqual(observed, [3,3])
```

## Po spuštění `test_factorize.py`:

```
...F...
======================================================================
FAIL: test_nine (__main__.FactorizeTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<wingdb_compile>", line 32, in test_nine
AssertionError: Lists differ: [9] != [3, 3]
[...snip...]
----------------------------------------------------------------------
Ran 7 tests in 0.000s
```

---

## Upravujeme `factorization.py`:

```
def factorize(multiple):
  factors = []
  for factor in range(2,multiple+1):
    while multiple % factor == 0:
      factors.append(factor)
      multiple /= factor
  return factors
```

# TDD Ukázka: Test faktorizace čísla 9

## Upravujeme `test_factorize.py`

```
  def test_nine(self):
    observed = factorize(9)
    self.assertEqual(observed, [3,3])
```

## Po spuštění `test_factorize.py`:

```
...F...
======================================================================
FAIL: test_nine (__main__.FactorizeTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<wingdb_compile>", line 32, in test_nine
AssertionError: Lists differ: [9] != [3, 3]
[...snip...]
----------------------------------------------------------------------
Ran 7 tests in 0.000s
```

---

## Upravujeme `factorization.py`:

```
def factorize(multiple):
  factors = []
  for factor in range(2,multiple+1):
    while multiple % factor == 0:
      factors.append(factor)
      multiple /= factor
  return factors
```

```
.......
----------------------------------------------------------------------
Ran 7 tests in 0.015s
```

# TDD Ukázka: Test faktorizace čísla 9

## Upravujeme `test_factorize.py`

```
  def test_nine(self):
    observed = factorize(9)
    self.assertEqual(observed, [3,3])
```

## Po spuštění `test_factorize.py`:

```
...F...
======================================================================
FAIL: test_nine (__main__.FactorizeTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<wingdb_compile>", line 32, in test_nine
AssertionError: Lists differ: [9] != [3, 3]
[...snip...]
----------------------------------------------------------------------
Ran 7 tests in 0.000s
```

---

## Upravujeme `factorization.py`:

```
def factorize(multiple):
  factors = []
  for factor in range(2,multiple+1):
    while multiple % factor == 0:
      factors.append(factor)
      multiple /= factor
  return factors
```

```
.......
----------------------------------------------------------------------
Ran 7 tests in 0.015s
```

---

- Jsme schopni přijít na nějaký další test, kde by náš kód selhal?

# TDD Ukázka: Test faktorizace čísla 9

Upravujeme `test_factorize.py`

```
  def test_nine(self):
    observed = factorize(9)
    self.assertEqual(observed, [3,3])
```

Po spuštění `test_factorize.py`:

```
...F...
======================================================================
FAIL: test_nine (__main__.FactorizeTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<wingdb_compile>", line 32, in test_nine
AssertionError: Lists differ: [9] != [3, 3]
[...snip...]
----------------------------------------------------------------------
Ran 7 tests in 0.000s
```

---

Upravujeme `factorization.py`:

```
def factorize(multiple):
  factors = []
  for factor in range(2,multiple+1):
    while multiple % factor == 0:
      factors.append(factor)
      multiple /= factor
  return factors
```

```
.......
----------------------------------------------------------------------
Ran 7 tests in 0.015s
```

---

- Jsme schopni přijít na nějaký další test, kde by náš kód selhal?
- Nevadí náhodou, že jako faktory bereme všechna čísla a nikoli jen prvočísla? Jak by se kód lišil?

# TDD: Závěr



Testy

- slouží jako specifikace.
- slouží jako dokumentace.
- pomáhají pochopit algoritmus.
- pomáhají předejít zbytečným složitostem v kódu.
- určují, kdy “je hotovo”.
- pomáhají zajistit, abychom úpravami do kódu nevnesli nové chyby.