AE4B99RPH: Problem Solving and Games
Automated Tests. Test-Driven Development.

Petr Pošík

Dept. of Cybernetics
CTU FEE

Motivation

Feedback from the first lab test

Task: Equip class `MyVector` with the following methods:

- ✔ `__add__(self, other)`: addition of 2 vectors
- ✔ `norm(self)`: the Euclidean norm (length) of the vector

Feedback from the first lab test

Task: Equip class `MyVector` with the following methods:

- ✔ `__add__(self, other)`: addition of 2 vectors
- ✔ `norm(self)`: the Euclidean norm (length) of the vector

Mistakes more common than expected:

- ✔ The submitted module was not named `vectors.py`.

Feedback from the first lab test

Task: Equip class `MyVector` with the following methods:

- ✔ `__add__(self, other)`: addition of 2 vectors
- ✔ `norm(self)`: the Euclidean norm (length) of the vector

Mistakes more common than expected:

- ✔ The submitted module was not named `vectors.py`.
- ✔ The class was not named `MyVector`.

Feedback from the first lab test

Task: Equip class `MyVector` with the following methods:

- ✔ `__add__(self, other)`: addition of 2 vectors
- ✔ `norm(self)`: the Euclidean norm (length) of the vector

Mistakes more common than expected:

- ✔ The submitted module was not named `vectors.py`.
- ✔ The class was not named `MyVector`.
- ✔ The methods were not named `__add__()` and `norm()`.

Feedback from the first lab test

Task: Equip class `MyVector` with the following methods:

- ✔ `__add__(self, other)`: addition of 2 vectors
- ✔ `norm(self)`: the Euclidean norm (length) of the vector

Mistakes more common than expected:

- ✔ The submitted module was not named `vectors.py`.
- ✔ The class was not named `MyVector`.
- ✔ The methods were not named `__add__()` and `norm()`.
- ✔ Method `__add__` did not return an instance of class `MyVector`.

Feedback from the first lab test

Task: Equip class `MyVector` with the following methods:

- ✔ `__add__(self, other)`: addition of 2 vectors
- ✔ `norm(self)`: the Euclidean norm (length) of the vector

Mistakes more common than expected:

- ✔ The submitted module was not named `vectors.py`.
- ✔ The class was not named `MyVector`.
- ✔ The methods were not named `__add__()` and `norm()`.
- ✔ Method `__add__` did not return an instance of class `MyVector`.
- ✔ The source code did not get the indentation (structure) right.

Trivial failures to fulfill the given specifications!

Feedback from the first lab test

Task: Equip class `MyVector` with the following methods:

- ✔ `__add__(self, other)`: addition of 2 vectors
- ✔ `norm(self)`: the Euclidean norm (length) of the vector

Mistakes more common than expected:

- ✔ The submitted module was not named `vectors.py`.
- ✔ The class was not named `MyVector`.
- ✔ The methods were not named `__add__()` and `norm()`.
- ✔ Method `__add__` did not return an instance of class `MyVector`.
- ✔ The source code did not get the indentation (structure) right.

Trivial failures to fulfill the given specifications!

Why did not you discover these bugs?

Feedback from the first lab test

Task: Equip class `MyVector` with the following methods:

- ✔ `__add__(self, other)`: addition of 2 vectors
- ✔ `norm(self)`: the Euclidean norm (length) of the vector

Mistakes more common than expected:

- ✔ The submitted module was not named `vectors.py`.
- ✔ The class was not named `MyVector`.
- ✔ The methods were not named `__add__()` and `norm()`.
- ✔ Method `__add__` did not return an instance of class `MyVector`.
- ✔ The source code did not get the indentation (structure) right.

Trivial failures to fulfill the given specifications!

Why did not you discover these bugs?

How to test your own code?

Motivation
Feedback from the first lab test
Test it in Python shell
Test it when you run the module
Test it with the help of testing tools

Testing

Test-Driven Development

Test it in Python shell

Run the Python shell and try to use the code as expected:

```
>>> 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]
```

Test it in Python shell

Run the Python shell and try to use the code as expected:

```
>>> 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]
```

- ✔ You would detect all the above mentioned mistakes.
- ✔ Sometimes you have to change the working directory (`import os; os.chdir()`).
- ✔ Issues with re-importing already imported module:
  - ✘ Python 2x: `reload(module)`
  - ✘ Python 3x: `import imp; imp.reload(module)`
  - ✘ …yet, it is not a good solution.
  - ✘ Reliable solution: restart the shell.

Test it when you run the module

Take advantage of `if __name__==’__main__’:` to run the tests:

```
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())
```

Test it when you run the module

Take advantage of `if __name__=='__main__':` to run the tests:

```
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())
```

- ✔ No need to worry about the working directory.
- ✔ Your “test” will work even without the explicit module import — if the module name is wrong, we can overlook it.
- ✔ However, the import can be used explicitely. It does not harm and the wrong module name will be discovered.

Test it with the help of testing tools

Modules for automated testing:

- ✔ doctest, unittest, or other frameworks
- ✔ you can run a lot of tests at once with all the results nicely summarized

Automated testing

Based on
Gerard Meszarosz: *xUnit Test Patterns: Refactoring Test Code*, Addison-Wesley, 2007.

Testing

Testing from the QA team point of view:

- ✔ Ensure that the code fulfills customer requirements and does not contain bugs.
- ✔ Test after the code is complete.
- ✔ The feedback is too late.

Testing

Testing from the QA team point of view:

- ✔ Ensure that the code fulfills customer requirements and does not contain bugs.
- ✔ Test after the code is complete.
- ✔ The feedback is too late.

Testing from the programmer’s point of view (unit tests, integration tests):

- ✔ Ensure that the unit I am working on right now fulfills the requirements that emerged as a result of the application architecture design.
- ✔ Test during development.
- ✔ The feedback comes much sooner.

Programmer’s testing

Hopefully you do at least some testing during development.

```
if __name__ == "__main__":
    pg = PrimesGenerator()
    print("Primes up to 0: ", pg.get_primes_up_to(0))
    print("Primes up to 1: ", pg.get_primes_up_to(1))
    print("Primes up to 2: ", pg.get_primes_up_to(2))
    print("Primes up to 3: ", pg.get_primes_up_to(3))
    print("Primes up to 4: ", pg.get_primes_up_to(4))
    print("Primes up to 5: ", pg.get_primes_up_to(5))
    print("Primes up to 6: ", pg.get_primes_up_to(6))
    print("Primes up to 20: ", pg.get_primes_up_to(20))
```

But you have to check the output:

```
Primes up to 0:  []
Primes up to 1:  []
Primes up to 2:  [2]
Primes up to 3:  [2, 3]
Primes up to 4:  [2, 3]
Primes up to 5:  [2, 3, 5]
Primes up to 6:  [2, 3, 5]
Primes up to 20:  [2, 3, 5, 7, 11, 13, 17, 19]
Primes up to 100:  [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 6
>>>
```

Automated tests: F.I.R.S.T.

Automated tests should be

Fast. If they are not fast, you will not run them often. If you will not run them often, you will not discover bugs in time.

Automated tests: F.I.R.S.T.

Automated tests should be

Fast. If they are not fast, you will not run them often. If you will not run them often, you will not discover bugs in time.

Independent. The test should be able to run in isolation and in any order. If they are not independent, a bug in a single test will trigger a series of bugs in other tests. Finding the bug will be harder.

Automated tests: F.I.R.S.T.

Automated tests should be

Fast. If they are not fast, you will not run them often. If you will not run them often, you will not discover bugs in time.

Independent. The test should be able to run in isolation and in any order. If they are not independent, a bug in a single test will trigger a series of bugs in other tests. Finding the bug will be harder.

Repeatable. Anybody should be able to repeat the tests anywhere with the same results.

Automated tests: F.I.R.S.T.

Automated tests should be

Fast. If they are not fast, you will not run them often. If you will not run them often, you will not discover bugs in time.

Independent. The test should be able to run in isolation and in any order. If they are not independent, a bug in a single test will trigger a series of bugs in other tests. Finding the bug will be harder.

Repeatable. Anybody should be able to repeat the tests anywhere with the same results.

Self-validating. The tests should either pass or fail. You shouldn’t be forced to parse some textual output of results to see if the test passed, otherwise you will not want to run the tests so often.

Motivation

Testing

Testing

Programmer's testing

Automated tests: F.I.R.S.T.

Doctest module

xUnit Framework

Test-Driven Development

Automated tests: F.I.R.S.T.

Automated tests should be

Fast. If they are not fast, you will not run them often. If you will not run them often, you will not discover bugs in time.

Independent. The test should be able to run in isolation and in any order. If they are not independent, a bug in a single test will trigger a series of bugs in other tests. Finding the bug will be harder.

Repeatable. Anybody should be able to repeat the tests anywhere with the same results.

Self-validating. The tests should either pass or fail. You shouldn't be forced to parse some textual output of results to see if the test passed, otherwise you will not want to run the tests so often.

Timely. The tests should be written in time, ideally before the production code. If you write them after the production code, the code is often hard to test. If writing the tests is hard, you will not want to write them.

Doctest module

- ✔ you have already seen it during lectures and labs
- ✔ special to Python (correct me if I am wrong)
- ✔ very handy for simple tests with little setup and cleanup, unnatural for more complex tests

```python
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(7)
    [2, 3, 5, 7]
    >>> pg.get_primes_up_to(20)
    [2, 3, 5, 7, 11, 13, 17, 19]
    """
    ...

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

xUnit Framework

- ✔ Standard unit testing framework
- ✔ Implemented in many languages (learn it once, use it anywhere)
- ✔ Python implementation: module `unittest`.

xUnit Framework

- ✔ Standard unit testing framework
- ✔ Implemented in many languages (learn it once, use it anywhere)
- ✔ Python implementation: module `unittest`.

```python
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()
```

Test-Driven Development

TDD: Test-Driven Development

Three rules of TDD:

1. Do not write any production code until you have first written a failing unit test.
2. Do not write more of a unit test than is sufficient to fail, and not compiling is failing.
3. Do not write more production code than is sufficient to pass the currently failing unit test.

TDD: Test-Driven Development

Three rules of TDD:

1. Do not write any production code until you have first written a failing unit test.
2. Do not write more of a unit test than is sufficient to fail, and not compiling is failing.
3. Do not write more production code than is sufficient to pass the currently failing unit test.

The result of these rules:

- ✔ a very short cycle in which you alternate between
  - ✘ the role of a customer who says *what* shall be done (you write a test), and
  - ✘ the role of a programmer who says *how* it shall be done (you write or modify production code).
- ✔ Tests and production code are written together (tests a few seconds sooner).
- ✔ Tests then cover the whole production code!

TDD Example

Create a function that factorizes a natural number into a product of prime factors.

- ✔ Input: the number to be factorized
- ✔ Output: a list of primes whose product is equal to the given number

TDD Example

Create a function that factorizes a natural number into a product of prime factors.

- ✔ Input: the number to be factorized
- ✔ Output: a list of primes whose product is equal to the given number

How would you proceed? Suppose we already have class `PrimeGenerator`...

TDD Example: Initial phase

Create the test file, `test_factorize.py`

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

TDD Example: Initial phase

Create the test file, `test_factorize.py`

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

After executing `test_factorize.py`:

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

TDD Example: Initial phase

Create the test file, `test_factorize.py`

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

After executing `test_factorize.py`:

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

Create an empty module, `factorization.py`

TDD Example: Initial phase

Create the test file, `test_factorize.py`

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

After executing `test_factorize.py`:

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

Create an empty module, `factorization.py`

After executing `test_factorize.py`:

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

TDD Example: Initial phase

Create the test file, `test_factorize.py`

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

After executing `test_factorize.py`:

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

Create an empty module, `factorization.py`

After executing `test_factorize.py`:

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

Update `factorization.py`:

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

TDD Example: Initial phase

Create the test file, test_factorize.py

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

After executing test_factorize.py:

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

Create an empty module, factorization.py

After executing test_factorize.py:

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

Update factorization.py:

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

After executing test_factorize.py:

```
--- Žádný výstup, kód bez chyby. ---
```

TDD Example: Initial phase

Create the test file, test_factorize.py

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

After executing test_factorize.py:

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

Create an empty module, factorization.py

After executing test_factorize.py:

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

Update factorization.py:

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

After executing test_factorize.py:

```
--- Žádný výstup, kód bez chyby. ---
```

Update test_factorize.py

```
import unittest
from factorization import factorize

class FactorizeTest(unittest.TestCase):
  pass

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

TDD Example: Initial phase

Create the test file, test_factorize.py

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

After executing test_factorize.py:

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

Create an empty module, factorization.py

After executing test_factorize.py:

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

Update factorization.py:

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

After executing test_factorize.py:

```
--- Žádný výstup, kód bez chyby. ---
```

Update test_factorize.py

```
import unittest
from factorization import factorize

class FactorizeTest(unittest.TestCase):
  pass

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

After executing test_factorize.py:

```

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

OK
builtins.SystemExit: False
```

TDD Example: Factorize number 2

Update `test_factorize.py`

```python
class FactorizeTest(unittest.TestCase):

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

TDD Example: Factorize number 2

Update test_factorize.py

```python
class FactorizeTest(unittest.TestCase):

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

After executing 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
```

Update factorization.py:

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

TDD Example: Factorize number 2

Update test_factorize.py

```
class FactorizeTest(unittest.TestCase):

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

After executing 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
```

Update 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 Example: Factorize number 2

Update test_factorize.py

```
class FactorizeTest(unittest.TestCase):

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

After executing 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
```

Update 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
```

Update factorization.py:

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

TDD Example: Factorize number 2

Update test_factorize.py

```
class FactorizeTest(unittest.TestCase):

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

After executing 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
```

Update 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
```

Update factorization.py:

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

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

TDD Example: Factorize number 3

Update `test_factorize.py`

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

TDD Example: Factorize number 3

Update test_factorize.py

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

After executing 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
```

Update factorization.py:

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

TDD Example: Factorize number 3

Update `test_factorize.py`

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

After executing `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
```

Update `factorization.py`:

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

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

TDD Example: Factorize number 4

Update `test_factorize.py`

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

TDD Example: Factorize number 4

Update test_factorize.py

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

After executing 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
```

Update factorization.py:

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

TDD Example: Factorize number 4

Update test_factorize.py

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

After executing 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
```

Update 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 Example: Factorize number 4

Update test_factorize.py

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

After executing 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
```

Update 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
```

Update factorization.py:

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

TDD Example: Factorize number 4

Update test_factorize.py

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

After executing 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
```

Update 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
```

Update 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 Example: Factorize number 5

Update `test_factorize.py`

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

TDD Example: Factorize number 5

Update `test_factorize.py`

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

After executing `test_factorize.py`:

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

TDD Example: Factorize number 6

Update `test_factorize.py`

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

TDD Example: Factorize number 6

Update `test_factorize.py`

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

After executing `test_factorize.py`:

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

TDD Example: Factorize number 6

Update test_factorize.py

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

After executing test_factorize.py:

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

Test for factorizing number 7 is left out, it is the same case as for numbers 3 and 5.

TDD Example: Factorize number 8

Update test_factorize.py

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

TDD Example: Factorize number 8

Update `test_factorize.py`

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

After executing `test_factorize.py`:

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

TDD Example: Factorize number 9

Update `test_factorize.py`

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

TDD Example: Factorize number 9

Update test_factorize.py

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

After executing 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 Example: Factorize number 9

Update test_factorize.py

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

After executing 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
```

Update 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 Example: Factorize number 9

Update test_factorize.py

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

After executing 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
```

Update 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 Example: Factorize number 9

Update test_factorize.py

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

After executing 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
```

Update 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
```

- ✔ Are you able to come up with another failing test?

TDD Example: Factorize number 9

Update `test_factorize.py`

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

After executing `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
```

Update `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
```

- ✔ Are you able to come up with another failing test?
- ✔ Note that we actually do not need any `PrimeGenerator`; if we decided to use it, the code may be more complex!

More complex case: class Game

Typical use of class Game:

```
>>> g = Game(playerA, playerB, payoff_matrix, n_iterations)
>>> g.run()
>>> g.get_players_payoffs()
```

What are the class responsibilities we want to test?

More complex case: class Game

Typical use of class Game:

```
>>> g = Game(playerA, playerB, payoff_matrix, n_iterations)
>>> g.run()
>>> g.get_players_payoffs()
```

What are the class responsibilities we want to test?

- ✔ Method `get_players_payoffs()` returns `(None,None)` before executing method `run()`.

More complex case: class Game

Typical use of class Game:

```
>>> g = Game(playerA, playerB, payoff_matrix, n_iterations)
>>> g.run()
>>> g.get_players_payoffs()
```

What are the class responsibilities we want to test?

- ✔ Method `get_players_payoffs()` returns `(None,None)` before executing method `run()`.
- ✔ Method `run()` calls methods `move()` and `record_opponents_move()` of both `Players` exactly `n_iterations` times.
- ✔ Method `run()` calls methods `move()` and `record_opponents_move()` alternatively, it begins with method `move()`.
- ✔ Method `run()` is fair to both the `Players`, i.e. it does not pass the current move of one player to the other player.
- ✔ …

TDD: Conclusions

Tests

- ✔ serve as specification by example.
- ✔ serve as documentation.
- ✔ help to understand the algorithm.
- ✔ help to prevent unnecessary complexity of the code.
- ✔ determine when we are "Done."
- ✔ help to prevent new bugs when modifying the code.