Unit Testing
or what can we learn from tech startups

2014-04-01
Saulius Lukauskas
Outline

• Motivation
• Hierarchy of Testing
• Short tutorial of unit testing in Python
• Two rules of writing testable code
• Kickstarting
This is the standard software workflow which most of people are familiar with.
Unfortunately, when people talk about coding only the coding and production parts are mentioned, while the testing part is often not spoken about.
The better we get at coding, the less problems there will be with our software.

There is this misconception in software development, an arrogance one may say, that good programmers make good code that has few problems.
The better we get at testing, the less problems there will be with our software.

In reality is that people who are good at testing, not at programming per se, make good software. Good programmers just make easier to test code.
This is due to the fact the development cycle often looks like this.
The cost of having a bug in your software increases with each step of the cycle.
The bugs at the coding stage are very easy to spot — the compiler often warns about them. They are also so easy to fix, most of us do not even consider them “bugs” any more.
Bugs at the quality assurance (testing) stage are a bit harder to spot, often require running your software for a good few minutes, checking the output carefully. Once the bug is spotted the debugging also takes a while. These are the real bugs in your software.
The bugs at the final stage are the worst ones. It often takes you hours, if not days to spot that something is wrong. If it ever gets spotted. Fixing the bug then again takes ages, as you need to dissect half of your software to track the error.
Unit testing aims to combine the testing and coding step into one lump. The main goal is to make sure the real bugs of your software are as easy to spot and to debug as things like “uninitialised variable” or an uncaught exception.
Example Problem
Chi-Square goodness-of-fit test for poisson distribution
Here’s the relatively easy-to-type data, taken from some tutorial from Plymouth uni

http://www.cimt.plymouth.ac.uk/projects/mepres/alevel/
fstats_ch5.pdf
My Implementation

```python
def poisson_chi_square(observed_data, total):
    ""
    Computes Pearson's Chi Square goodness-of-fit score for the observed data, 
given the bins to put the data in, assuming poisson distribution 

    Puts the data that cannot be binned in the bins into a special "other" bin.
    ""
    >>> poisson_chi_square({1: 1, 2: 2, 3:3, 'other': 1}, total=15)
    2.8703641972054568

    :param observed_data: the observed data
    :param total: total number of particles (required for lambda estimation 
                 as "other" bin is ambiguous)
    ""
```

Docstring for the implementation describing how to use the function.

NB: The highlighted code example above is called a doctest and can be used to ensure your documentation stays relevant to code.
from math import factorial, exp

SAMPLE_DATA = {0: 77, 1: 98, 2: 55, 3: 30, 'other': 0}
SAMPLE_TOTAL = 325

def poisson_chisquare(observed_data, total):
    # Compute the poisson parameter from the data
    n = sum(observed_data.values())
    lambda_estimate = float(total) / n

    probabilities_of_bins = {}
    for bin in observed_data:
        if bin == 'other':
            continue  # Deal with this later
        p_i = (((lambda_estimate ** bin) / factorial(bin)) * exp(-lambda_estimate))
        probabilities_of_bins[bin] = n * p_i

    probabilities_of_bins['other'] = 1 - sum(probabilities_of_bins.values())

    expected_counts = {}  # expected count for each bin
    for bin, p_i in probabilities_of_bins.items():
        expected_counts[bin] = n * p_i

    chi_square = 0
    for bin, observed_count in observed_data.items():
        expected_count = expected_counts[bin]
        chi_square_increment = (expected_count - observed_count)**2 / expected_count
        chi_square += chi_square_increment

    return chi_square
How should we know if the code is correct?

Any suggestions?
Well, we already have a test for this function, it is encoded in the data.
The red area is the input, the green area is the output.
This is called a scenario test

Dataset

 ![Image of a table and a red and green area]

<table>
<thead>
<tr>
<th>$x_i$</th>
<th>$O_i = f_i$</th>
<th>$P(X = x_i)$</th>
<th>$E_i$</th>
<th>$(O_i - E_i)$</th>
<th>$(O_i - E_i)^2$</th>
<th>$\frac{(O_i - E_i)^2}{E_i}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>77</td>
<td>0.2865</td>
<td>74.5</td>
<td>2.5</td>
<td>6.25</td>
<td>0.084</td>
</tr>
<tr>
<td>1</td>
<td>90</td>
<td>0.3581</td>
<td>93.1</td>
<td>-3.1</td>
<td>9.61</td>
<td>0.103</td>
</tr>
<tr>
<td>2</td>
<td>55</td>
<td>0.2238</td>
<td>58.2</td>
<td>-3.2</td>
<td>10.24</td>
<td>0.176</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0.0933</td>
<td>24.2</td>
<td>5.8</td>
<td>33.64</td>
<td>1.390</td>
</tr>
<tr>
<td>≥4</td>
<td>8</td>
<td>0.0383</td>
<td>10.0</td>
<td>-2.0</td>
<td>4.00</td>
<td>0.400</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.0000</td>
<td></td>
<td></td>
<td></td>
<td>2.153</td>
</tr>
</tbody>
</table>
Hierarchy of Testing

Scenario Tests

Test your **whole** system for some particular user behaviour
Example Scenario Test

Given the input from a couple of slides ago, and assuming a Poisson distribution, the resulting Chi-Square value should be 2.153
Some Typing:
In [2]: SAMPLE_DATA
Out[2]: {0: 77, 1: 90, 2: 55, 3: 30, 'other': 8}

Some more typing
In [3]: SAMPLE_TOTAL
Out[3]: 325

More typing, getting the result
In [4]: poisson_chi_square(SAMPLE_DATA, SAMPLE_TOTAL)
Out[4]: 2.1105516171151155

Verification: It’s not 2.153 :(

Everybody does scenario tests, we can cut down the number of typing every time we want to change this by making a script to run this.. OR.. A TEST
Quick Tutorial on Writing Tests in Python

```python
import unittest

from chi_square import *

SAMPLE_DATA = {0: 77, 1: 90, 2: 55, 3: 30, 'other': 0}
SAMPLE_TOTAL = 325

class ScenarioTests(unittest.TestCase):

def test_sample_data(self):
    """
    Given the sample_data and sample_total
    the function `poisson_chi_square`
    should return the answer 2.153
    """
    expected_answer = 2.153
    actual_answer = poisson_chi_square(SAMPLE_DATA, SAMPLE_TOTAL)

    self.assertEqual(expected_answer, actual_answer)
```
1. create new file, name it `test_<something>.py`

```python
import unittest
from chi_square import *

SAMPLE_DATA = {0: 77, 1: 90, 2: 55, 3: 30, 'other': 8}
SAMPLE_TOTAL = 325

class ScenarioTests(unittest.TestCase):
    def test_sample_data(self):
        """Given the sample_data and sample_total
        the function 'poisson_chi_square'
        should return the answer 2.153
        """
        expected_answer = 2.153
        actual_answer = poisson_chi_square(SAMPLE_DATA, SAMPLE_TOTAL)
        
        self.assertEqual(expected_answer, actual_answer)
```
2. import unittest package

```python
import unittest
from chi_square import *

SAMPLE_DATA = {0: 77, 1: 90, 2: 55, 3: 30, 'other': 0}
SAMPLE_TOTAL = 325

class ScenarioTests(unittest.TestCase):
    def test_sample_data(self):
        """
        Given the sample_data and sample_total
        the function 'poisson_chi_square'
        should return the answer 2.153
        """
        expected_answer = 2.153
        actual_answer = poisson_chi_square(SAMPLE_DATA, SAMPLE_TOTAL)

        self.assertEqual(expected_answer, actual_answer)
```
BTW,
this is not just a Python thing

- Python: built in `unittest module` + `nose`
- R: RUnit package
- C: At least a couple: Check, CUnit
- Java: JUnit
- MATLAB: Built in unit testing framework
```python
import unittest
from chi_square import *

SAMPLE_DATA = {0: 77, 1: 90, 2: 55, 3: 30, 'other': 8}
SAMPLE_TOTAL = 325

class ScenarioTests(unittest.TestCase):
    def test_sample_data(self):
        """
        Given the sample_data and sample_total
        the function 'poisson_chi_square'
        should return the answer 2.153
        """
        expected_answer = 2.153
        actual_answer = poisson_chi_square(SAMPLE_DATA, SAMPLE_TOTAL)
        self.assertEqual(expected_answer, actual_answer)
```
3. import your function

```python
import unittest
from chi_square import *

SAMPLE_DATA = {0: 77, 1: 90, 2: 55, 3: 30, 'other': 0}
SAMPLE_TOTAL = 325

class ScenarioTests(unittest.TestCase):
    def test_sample_data(self):
        
        Given the sample_data and sample_total
        the function `poisson_chi_square`
        should return the answer 2.153
        
        expected_answer = 2.153
        actual_answer = poisson_chi_square(SAMPLE_DATA, SAMPLE_TOTAL)

        self.assertEqual(expected_answer, actual_answer)
```
4. Make a class that inherits from unittest.TestCase

```python
import unittest
from chi_square import *

SAMPLE_DATA = {0: 77, 1: 90, 2: 55, 3: 30, 'other': 0}
SAMPLE_TOTAL = 325

class ScenarioTests(unittest.TestCase):
    def test_sample_data(self):
        """Given the sample_data and sample_total the function 'poisson_chi_square' should return the answer 2.153"""
        expected_answer = 2.153
        actual_answer = poisson_chi_square(SAMPLE_DATA, SAMPLE_TOTAL)

        self.assertEqual(expected_answer, actual_answer)
```
5. Create a test method in the class, comment it properly.

```python
import unittest
from chi_square import *

SAMPLE_DATA = {0: 77, 1: 90, 2: 55, 3: 30, 'other': 0}
SAMPLE_TOTAL = 325

class ScenarioTests(unittest.TestCase):
    def test_sample_data(self):
        """
        Given the sample_data and sample_total
        the function `poisson_chi_square`
        should return the answer 2.153
        """
        expected_answer = 2.153
        actual_answer = poisson_chi_square(SAMPLE_DATA, SAMPLE_TOTAL)
        self.assertEqual(expected_answer, actual_answer)
```

The Given <...> the <...> should <..> template is really good one to use to document tests.
6. Get the actual answer, and compare it to expected one

```python
import unittest
from chi_square import *

SAMPLE_DATA = {0: 77, 1: 90, 2: 55, 3: 30, 'other': 8}
SAMPLE_TOTAL = 325

class ScenarioTests(unittest.TestCase):
    def test_sample_data(self):
        """
        Given the sample_data and sample_total
        the function `poisson_chisquare`
        should return the answer 2.153
        """
        expected_answer = 2.153
        actual_answer = poisson_chisquare(SAMPLE_DATA, SAMPLE_TOTAL)
        self.assertEqual(expected_answer, actual_answer)
```
7. Install “nose” package, and run tests

```
  presentations (master *)$ pip install nose
  Requirement already satisfied (use --upgrade to upgrade): nose in /Library/Python/2.7/site-packages/nose-1.2.1-py2.7.egg
  Cleaning up...
  presentations (master *)$ nosetests
  FAIL: Given the sample_data and sample_total

  Traceback (most recent call last):
    File "~/Users/saulius/dev/presentations/unit-testing/presentations/test_chisquare.py", line 18, in test_sample_data
    self.assertEqual(expected_answer, actual_answer)
  AssertionError: 2.1531 != 2.1105516171151155

  Ran 1 test in 0.006s
  FAILED (failures=1)
```

Nose package is not strictly necessary, but makes your life easier
Same result as before

Traceback (most recent call last):
  File "/Users/saulius/dev/presentations/unit-test-runner.py", line 18, in test_sample_data
    self.assertEqual(expected_answer, actual_answer)
AssertionError: 2.153 != 2.1105516171151155
Same result as before

```
Traceback (most recent call last):
  File "/Users/saulius/dev/presentations/unit-testare.py", line 18, in test_sample_data
    self.assertEquals(expected_answer, actual_answer)
AssertionError: 2.153 != 2.1105516171151155
```

except:
  • fully automated (no more typing of inputs in the shell)
  • easily reproducible
  • will live as long as your code does
2.1105516171151155
!=
2.153

What happened?
Scenario tests tell us *if* something failed, not *where* the problem is.

.. To get the answer to where the problem lies we need to dig deeper to the world of functional tests.
Hierarchy of Testing

Scenario Tests

Functional Tests
Test a part of your system responsible for a certain function in isolation

Functional tests isolate parts of your system and test them separately
You do functional tests already

<table>
<thead>
<tr>
<th>$x_i$</th>
<th>$O_i$</th>
<th>$P(X=x_i)$</th>
<th>$E_i$</th>
<th>$(O_i - E_i)$</th>
<th>$(O_i - E_i)^2$</th>
<th>$\frac{(O_i - E_i)^2}{E_i}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>77</td>
<td>0.2865</td>
<td>74.5</td>
<td>2.5</td>
<td>6.25</td>
<td>0.084</td>
</tr>
<tr>
<td>1</td>
<td>90</td>
<td>0.3581</td>
<td>93.1</td>
<td>-3.1</td>
<td>9.61</td>
<td>0.103</td>
</tr>
<tr>
<td>2</td>
<td>55</td>
<td>0.2238</td>
<td>68.2</td>
<td>-3.2</td>
<td>10.24</td>
<td>0.176</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0.0933</td>
<td>24.2</td>
<td>5.8</td>
<td>33.64</td>
<td>1.390</td>
</tr>
<tr>
<td>≥4</td>
<td>8</td>
<td>0.0383</td>
<td>10.0</td>
<td>-2.0</td>
<td>4.00</td>
<td>0.400</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.0000</td>
<td></td>
<td></td>
<td></td>
<td>2.153</td>
</tr>
</tbody>
</table>

http://www.cimt.plymouth.ac.uk/projects/mepres/alevel/fstats_ch5.pdf

Functional test #1
You do functional tests already

<table>
<thead>
<tr>
<th>$x_i$</th>
<th>$O_i = f_i$</th>
<th>$P(X = x_i)$</th>
<th>$E_i$</th>
<th>$O_i - E_i$</th>
<th>$(O_i - E_i)^2$</th>
<th>$\frac{(O_i - E_i)^2}{E_i}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>77</td>
<td>0.2865</td>
<td>74.5</td>
<td>2.5</td>
<td>6.25</td>
<td>0.084</td>
</tr>
<tr>
<td>1</td>
<td>90</td>
<td>0.3581</td>
<td>93.1</td>
<td>-3.1</td>
<td>9.61</td>
<td>0.103</td>
</tr>
<tr>
<td>2</td>
<td>55</td>
<td>0.2238</td>
<td>58.2</td>
<td>-3.2</td>
<td>10.24</td>
<td>0.176</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0.0933</td>
<td>24.2</td>
<td>5.8</td>
<td>33.64</td>
<td>1.390</td>
</tr>
<tr>
<td>≥4</td>
<td>8</td>
<td>0.0383</td>
<td>10.0</td>
<td>-2.0</td>
<td>4.00</td>
<td>0.400</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Output: $2.153$

http://www.cimt.plymouth.ac.uk/projects/mepres/alevel/fstats_ch5.pdf

Functional test #2
Example Functional Tests

• Given that the input is as specified in the table a couple of slides ago, the probabilities of each of the bins should be equal to the ones specified in the same table.

• Given that the expected counts are as specified in the table, and observed counts are as specified in the table, the Chi-Square score should be 2.153
With this code, we cannot isolate the two parts of the system and test them separately — they are lumped together.
As a side effect, `chi_square_score` becomes reusable for all distributions.

Note how Chi-Square function suddenly becomes reusable.
def test_expected_counts_calculation_using_sample_data(self):
    """
    Given sample data and sample total,
    the function compute_expected_counts_poisson
    should return the expected counts that should be
    same as the ones in the table, up to 1 decimal place difference
    """
    expected_answer = {0: 74.5, 1: 93.1, 2: 58.2, 3: 24.2, 'other': 10}
    actual_answer = compute_expected_counts_poisson(SAMPLE_DATA, SAMPLE_TOTAL)
    # Check if the same set of dictionary keys were returned:
    self.assertEqual(expected_answer.keys(), actual_answer.keys())
    # Check if all elements of the dictionaries are equal to 1 decimal place
    for key, value in expected_answer.items():
        self.assertAlmostEqual(value, actual_answer[key], delta=0.1)
Adding Functional Tests (II)

def test_chisquare_value_calculation_using_sample_data(self):
    
    Given the sample expected counts, and sample observed counts,
    the function chi_square_score
    should return the chi_square score as described in the table
    
    expected_counts = {0: 74.5, 1: 93.1, 2: 58.2, 3: 24.2, 'other': 10}

    expected_answer = 2.153
    actual_answer = chi_square_score(SAMPLE_DATA, expected_counts)
    self.assertEqual(expected_answer, actual_answer)
Run Tests
(this time from PyCharm)
Run Tests
(this time from PyCharm)
Failure
Traceback (most recent call last):
  File "/Users/saulius/dev/presentations/unit-testing/presentations/test_chi_square.py", line 47, in test_chi_square_value_calculation_using_sample_data
    self.assertEqual(expected_answer, actual_answer)
AssertionError: 2.153 != 1.753142620828101
Functional tests tell us *where* something fails, not *what* fails
Hierarchy of Testing

Scenario Tests
  Functional Tests
  Unit Tests
  Test a small unit of your system in isolation
You do unit tests already

\[
\begin{array}{cccccc}
\times_i & O_i = f_i & P(X = x_i) & E_i & (O_i - E_i) & \frac{(O_i - E_i)^2}{E_i} \\
0 & 77 & 0.2865 & 74.5 & 2.5 & 6.25 & 0.084 \\
1 & 90 & 0.3581 & 93.1 & -3.1 & 9.61 & 0.103 \\
2 & 55 & 0.2238 & 58.2 & -3.2 & 10.24 & 0.176 \\
3 & 30 & 0.0933 & 24.2 & 5.8 & 33.64 & 1.390 \\
\geq 4 & 8 & 0.0383 & 10.0 & -2.0 & 4.00 & 0.400 \\
\hline
& & & & 1.0000 & 2.153 & \\
\end{array}
\]

http://www.cimt.plymouth.ac.uk/projects/mepres/alevel/fstats_ch5.pdf
Example Unit Tests

• Given that total number of observations is 325, and the number of distinct time points sampled is 260, the estimated lambda should be $260/325 = 0.8$

• Given that lambda=0.8, and $k=1$, the probability $P(X=k|\lambda)$ should be equal to $\lambda^k / k! * e^{-\lambda}$

• Given that observed count is equal to 8 and the expected count is equal to 10, the Chi-Square increment should be $(8-10)^2/10$
Preparing code for unit tests

def chi_square_coefficient(expected_count, observed_count):
    return (expected_count - observed_count) ** 2 / expected_count

def chi_square_score(observations, expected_counts):
    chi_square = 0
    for bin, observed_count in observations.items():
        expected_count = expected_counts[bin]
        chi_square_increment = chi_square_coefficient(expected_count, observed_count)
        chi_square += chi_square_increment
    return chi_square
def test_chisquare_coefficient(self):
    
    Given observed counts as 77, 90 and 8 and respective expected counts 74.5, 93.1, 10, the chisquare_coefficient function should return
    0.004, 0.103 and 0.400 respectively following from \( (O_i - E_i)^2 / E_i \) equation.
    
    self.assertEqual(0.004, chisquare_coefficient(74.5, 77), delta=1e-3)
    self.assertEqual(0.103, chisquare_coefficient(93.1, 90), delta=1e-3)
    self.assertEqual(0.400, chisquare_coefficient(10, 8), delta=1e-3)
Failure
Traceback (most recent call last):
  File "/Users/saulius/dev/presentations/unit-testing/presentations/test_chi_square.py", line 59, in test_chi_square_coefficient
    self.assertAlmostEqual(0.400, chi_square_coefficient(10, 3), delta=1e-3)
AssertionError: 0.4 != 0 within 0.001 delta

```python
def test_chi_square_coefficient(self):
    """
    Given observed counts as 77, 90 and 8 and respective expected counts 74.5, 93.1, 10,
    the chi_square_coefficient function should return
    0.004, 0.103 and 0.400 respectively following from (Oi - Ei)^2 / Ei equation.
    """
    self.assertAlmostEqual(0.004, chi_square_coefficient(74.5, 77), delta=1e-3)
    self.assertAlmostEqual(0.103, chi_square_coefficient(93.1, 90), delta=1e-3)
    self.assertAlmostEqual(0.400, chi_square_coefficient(10, 8), delta=1e-3)
```
Failure
Traceback (most recent call last):
  File "/Users/saulius/dev/presentations/unit-testing/presentations/test_chi_square.py", line 59, in test_chi_square_coefficient
    self.assertAlmostEqual(0.400, chi_square_coefficient(10, 8), delta=1e-3)
AssertionError: 0.4 != 0 within 0.001 delta

Can you guess what the problem is?
Fixing Integer Division

def chi_square_coefficient(expected_count, observed_count):
    return (expected_count - observed_count) ** 2 / float(expected_count)
Unit tests tell us “what” failed, not “why”
Sometimes the tests are wrong

Failure
Traceback (most recent call last):
  File "/Users/saulius/dev/presentations/unit-testing/presentations/test_chi_square.py", line 47, in test_chi_square_value_calculation_using_sample_data
    self.assertEqual(expected_answer, actual_answer)
AssertionError: 2.153 != 2.153142620828101

This should not be a failed test! (Use self.assertAlmostEqual)
Sometimes the tests are wrong

Given that these two functional tests work, but the first scenario test fails, we could also strongly believe that the scenario test is broken, and system is otherwise fine.

Broken tests should be fixed (if relevant) or removed (if not relevant).

This should not be a failed test!

And by the way, the scenario test is still failing. Given that our separate functions of the code work as expected as shown by the functional tests, we can be reasonably confident that scenario test is broken (i.e. due to lack of floating point precision in the table), and we should either remove it or fix it.
The Two Rules of Writing Testable code
class CoffeeMachine(object):
    def __init__(self):
        self.water_tank = WaterTank(volume=10)

cm = CoffeeMachine()
Dependency Injection
Classes should not create new objects

class CoffeeMachine(object):
    def __init__(self):
        self.water_tank = WaterTank(volume=10)

cm = CoffeeMachine()

Can you really believe that a coffee machine knows how to construct a water tank from scratch?

Imagine how painful testing whether the coffee machine shows an error message when the tank is empty, if the only way to drain the coffee tank was by making coffee and the volume is in litres. Would you do that in real life or just pour the water out?
Dependency Injection
Classes should not create new objects

class CoffeeMachine(object):
    def __init__(self, water_tank):
        self.water_tank = water_tank

    cm = CoffeeMachine(WaterTank(volume=10))

Factory knows how to construct a water tank and connect it to the coffee machine

Here we could just replace the water tank to something of smaller volume in the test and we are happy.
def save(filename, object):
    with open(filename, 'w') as file_:
        file_.write(serialise(object))

Here we ask for the filename,
but only need the file object
Imagine testing this on a
system without “write” permission
in the current directory
Law of Demeter

“Ask only for what you need”

def save(file_, object):
    file_.write(serialise(object))

If we ask for file only, in the test we could replace it with some other object that has write functionality

Similarly if you want to check the fluid level of the car, would you need to pass in the whole car, so you can just check the engine under the hood and fluid meter there, or is it better just the pass the fluid meter itself?
We have seen that writing tests make your code more modular.
There is an extreme way of writing unit tests, called Test-Driven Development (TDD).
During TDD, you write tests before writing code and make sure they pass.
Common belief is that this helps to write unit tests, but in fact it helps you to make your code structured better.
Kickstarting

Next time you are writing a new function write a test for it

However, TDD might be too much of a jump to get started with unit tests.
I suggest you to start by writing a test to the next function of your code you are writing.
Kickstarting

Next time you find a bug in your program, spend the 5 minutes writing a test for it.

And the next time you find a bug in program, write a test exploiting that bug.
Then fix the bug, and observe the tests going green.
Summary

• Automated tests help you speed the feedback loop of software development

• There are roughly three levels of the test hierarchy: Scenario, Functional and Unit Tests

• Testable code is also easier to maintain code

• Writing tests is easy, writing testable code is hard and takes practice
Recommended Watching

Miško Hevery “Psychology of Testing”
https://www.youtube.com/watch?v=pqomi6W4AJ4