Introduction to Software Engineering (ISAD1000)

Lecture 6: Testing

Updated: 16th February, 2022

Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

Outline

Nobody is Perfect

- Regardless of expertise, software engineers make mistakes.
- Any large software system is virtually *guaranteed* to have mistakes in it.
- However, it pays to find and fix as many as you can.
- Testing is not about finding compiler/syntax errors.
  - These are easy to find – just run the compiler!
- Testing is about finding logic errors – code that is *syntactically* correct, but *logically* wrong.
  - It does something, but not the right thing!
  - The compiler will just assume you know what you're doing.

Logic Errors

- Example syntax errors:

```
celcius temp = (fahrenheit temp - 32.0) / 1.8;
```

  - You can't have spaces in variable names.
  - The compiler will stop you.
- Example logic error:

```
celcius = (fahrenheit - 32.0) * 1.8;
```

  - Notice the "*" (multiplication). This is the wrong calculation.
  - The compiler *won't* stop you. It assumes you *meant* to write that. How would it know any better?
- Syntax errors are nothing, and logic errors are everything.
  - Both can be irritating and confusing.
  - But you can *guarantee* that all syntax errors have been fixed.
  - You won't even *know* about logic errors... unless you test or inspect your code!

Testing in Software Projects

- Testing is pervasive in software engineering.
- *Everything* is tested, from individual submodules to systems of millions of lines of code.
- There are many phases of testing:
  - Unit testing (small-scale) – most of this lecture;
  - Integration testing (medium-scale);
  - System testing (large-scale);
    - Function testing,
    - Performance testing,
    - Acceptance testing,
    - Installation testing;
  - Regression testing (after modification).
- There are many strategies for testing – we'll look at some basic ones.

Faults and Failures

- "Logic error" can be a slightly sloppy term.
- To be more precise, we distinguish between "faults" and "failures".
- Fault (or defect, or bug):
  - Mistakes or deficiencies in the code (or in relevant documents).
  - e.g. The coder wrote "$*$" rather than "$/$".
- Failure:
  - An event where the software fails to achieve its goals.
  - e.g. The program prints the wrong "celcius" value.
  - Also includes crashes, freezes, or any other incorrect behaviour.
- A fault has the potential to cause failures, but:
  - One fault can cause several failures.
  - One fault may (due to good luck) never cause any failures.
  - It sometimes takes *several* faults in combination to cause a failure.

Test Failures

- The purpose of testing is to cause failures.
- From failures, you can backtrack to find faults, and then fix them.
  - The finding-and-fixing part is called *debugging*.
  - Can't be done unless you know there's something wrong!
- Testing should trigger as many failures as possible.
- Then you can *fix* as many faults as possible.

Unit Testing

- We'll focus on unit testing.
  - (But remember that there are other forms of testing too.)
- Importantly, unit testing is not just "running your program to see what happens".
- You must design, implement and execute *test cases*.
- A test case is a separate piece of code – a separate submodule.
- It verifies that the "real" code works.

Production Code vs Test Code

- To avoid confusion, we make this distinction:
  - *Production code* is the source code that makes up the actual software system.
  - *Test code* is the source code that makes up test cases.
- These are always separated into different files.
  - Never mix production code and test code in the same file.
- The amount of test code often equals (or even exceeds) the amount of production code!
  - Software engineers spend a lot of time designing and writing test cases.
  - However, quality should be considered before quantity.

The need for test code

- Do you need test code?
- Can't you run the program normally to see if it works?
- Yes, you can. But can you do it 100, 1000, etc. times?
  - Each time, you must make the software do *everything* it is designed to do.
  - Imagine doing this on a large system.
- Why so much repetition?
  - You will be constantly changing your software.
  - You could make a mistake at any time.
  - To find faults quickly, you must test frequently.
  - Finding and fixing faults quickly will reduce their damage.
- Without test code, you may even forget what kinds of tests you need to perform.

Repeatability

- Most importantly, testing must be *repeatable*.
- A test case must have the same outcome every time, until you change the software.
- That means that if:
  1. your testing failed, and
  2. you then changed the production code (only), and
  3. your testing now passes,

  then you know you fixed the fault.
- If testing is not repeatable, this reasoning doesn't work.

Automation

- Automation is the best (simplest) way to make test cases repeatable.
  - Test cases run without any sort of user intervention.
  - (But you still need to manually create them!)
- All the information required by a test case can be "hard-coded" – embedded directly in the test code.
- User input is an unknown variable that we don't need here.
- Automation obviously also makes testing much less painful.

The need for *multiple* test cases

- Do you need more than one test case?
- Can't you put all test code inside a single test case? (That's simpler, right?)
- Yes, you can, but it will be practically useless.
- Each test case has only two possible outcomes:

  fail if the software doesn't perform as required, or
  pass if it does.
- With multiple test cases, you know *which one* (if any) has failed.
- This isolates the fault, making it much easier to find and fix.
- Multiple test cases are also much easier to write and modify.

Test-Driven Development (TDD)

- In TDD, the creation of test cases comes first.
- Your test cases embody the software requirements.
  - They describe exactly what the software must do.
- This is the job of the SRS (software requirements specification), but:
  - The SRS is written in natural language.
  - The SRS cannot automatically verify the software.
- TDD is how to keep a project on track without an SRS; i.e. in agile methods.

Test Design – General Concept

Tests must be *designed* before they can be coded:

1. List your test cases.
    - Production code usually has different logic for different situations. We want to test all of it.
    - Each test case should cover a different situation.
2. Pick **test data** (for each test case).
    - That thing you're testing? It (probably) needs some sort of input or import data.
    - To test it, you have to give it some data.
    - Choose the test data so that you're testing the "right thing".
3. Calculate **expected results** (for each test case).
    - A particular test data *should* give you a particular result.
    - What should that result be? (If you don't know, you can't verify it!)

Black Box and White Box Approaches

How do we know what test cases we need?

Black Box (you can't see inside the production code)

- Tests are designed based on the submodule specification alone – *what* they are supposed to do, but not their code.

White Box / Clear Box (you *can* see the production code)

- Tests are designed by analysing the "paths" through the production code.
- Advantage: with better knowledge of the production code, you may pick up more defects.
- Disadvantage: your test code is more likely to need updating if/when the production code changes.
- (We'll come back to this in a later lecture.)

You Can't Test Everything

- You cannot test your software with *every possible input*.
  - A 32-bit integer has 4 billion possible values.
  - *Two* 32-bit integers together have 18 *quintillion* possible values (1.8×10^{19}).
- These are the *small* cases!
- Instead, it's a balancing act.
- On one hand, we need a small-ish set of test cases.
- On the other hand, we want these to "cover" as much of the production code as possible.
  - Even if we can't test every input, we'd like to try to test every part of the *code*.
  - (Even if we can't actually see that code.)
- There are no guarantees, though.
  - Faults *will* still slip through from time to time.

Equivalence Partitioning (A Black Box Approach)

- Equivalence partitioning is a way of developing test cases.
- It works on two assumptions:
  1. Production code does different things based on its input data; i.e. it has different "categories" of behaviour.
  2. If one category of behaviour works once, it should work all the time.
- There is guesswork involved, so these assumptions are NOT always absolutely right, but it's a starting point.
- We figure out what these categories are, based on descriptions of the production code.
- We develop one test case per category. For each one:
  - We pick test data that should produce that category of behaviour.
  - We calculate the expected result(s).

Sgn Example

- Say we want to test this submodule (written by a colleague):

> Submodule **sgn**
> Imports: **n (real)**
> Exports: **result (integer)**
> Implements the "sign function". Returns -1, 0 or 1, if n is negative, 0 or positive, respectively.

- We can identify three categories of behaviour.
- We'll label the categories according to how they can be reproduced:
  1. $n < 0$
  2. $n = 0$
  3. $n > 0$
- These represent our test cases.
- Now we pick test data and expected results for each one.

Sgn Example (2)

It's easiest to show this in a table:

Category	Test Data	Expected Result
$n < 0$	$n = -5$	-1
$n = 0$	$n = 0$	0
$n > 0$	$n = 8$	1

- Each row is a separate test case.
- For test data, pick a value that conforms to each category.
- For the expected result, calculate (manually) what the production code should do, for that test data.

Results and Categories

- The categories are NOT ONLY about the export/result value.
- Consider a different production code submodule:

> Submodule **abs**
> Imports: **n (real)**
> Exports: **result (real)**
> Implements the absolute value function. If n is non-negative, returns n. Otherwise, returns the inverse of n.

- There are two distinct categories: $n \geq 0$ and $n < 0$.
- We can't tell this from the export value, though.

Category	Test Data	Expected Result
$n < 0$	$n = -10$	10
$n \geq 0$	$n = 10$	10

Other Data Types

- Our data may be non-numeric: Submodule **palindrome**
  Imports: **s (string)**
  Exports: **result (boolean)**
  Checks whether s is a palindrome; i.e. if it reads the same forwards and backwards. Returns true if it is, or false otherwise.
- We can't use <, ≤, etc., but there are still distinct behaviours.
- We can still design test cases:

Category	Test Data	Expected Result
s *is* a palindrome	`"glenelg"`	`true`
s *isn't* a palindrome	`"albuquerque"`	`false`

Multiple Imports

- Submodules frequently have multiple imports; e.g.:

> Submodule **max**
> Imports: **value1, value2 (integers)**
> Exports: **maximum (integer)**
> Determines the highest out of value1 and value2, and returns it.

- We consider imports *in combination* (not separately).
- We'd infer three categories of behaviour: value1 < value2, value1 = value2, and value1 > value2.
- And so…

Category	Test Data	Expected Result
value1 < value2	10, 20	20
value1 = value2	10, 10	10
value1 > value2	10, 5	10

Error Categories

- Production code must often perform error-handling:

> Submodule **formatTime**
> Imports: **inHours, inMins (integers)**
> Exports: **outTime (string)**
> Generates a string containing the time in 24-hour "HH:MM" format. Returns the string "error" if either inHours or inMins are invalid.

- Even if the code works perfectly, the *outside world* does not.
- Say formatTime is given invalid `inHours` or `inMins` values.
  - Not formatTime's fault! It can't control the data it receives.
  - But it still has to deal with it sensibly.
- If the production code performs error-handling, we must *test* that error handling.
  - Give it invalid values to see if it *correctly* reports an error.

Error Categories (2)

- So, does formatTime simply have *two* categories – valid and invalid?
- Let's be careful what we mean by a "category of behaviour".
  - We should test all the things the production code must do.
  - Including all the different error checks it must perform.
- How many *different ways* can "formatTime" return an error?
- How about *eight!* (Plus one for the valid case.)
  - Either `inHours` or `inMins` could be invalid, or both.
  - Either one could be either *too low*, in-range, or *too high*.
  - The production code is likely to check these things separately.
  - To catch bugs, we want to test all combinations.

Error Categories (3)

	Category inHours, inMins	Test data inHours, inMins	Expected Result
1	0–23, 0–59	12, 30	"12:30"
2	0–23, < 0	12, -10	"error"
3	0–23, ≥ 60	12, 70	"error"
4	< 0, 0–59	-3, 25	"error"
5	< 0, < 0	-3, -10	"error"
6	< 0, ≥ 60	-3, 70	"error"
7	≥ 24, 0–59	27, 25	"error"
8	≥ 24, < 0	27, -10	"error"
9	≥ 24, ≥ 60	27, 70	"error"

- One test for valid import values.
- Eight tests for various combinations of invalid values.

Categories as Ranges of Numbers

- In general, categories don't have a special notation.
  - You can always use plain English, and often you have to.
  - Provided it's *clear* and makes sense!
- But, for ranges of numbers, using $<$, $\leq$, etc. may be easier.
  - These are the mathematical "inequality" symbols.
  - But you have to get them right!
  - We often chain them: "$0 \leq x < 24$".
    - i.e. 0 is less-than-or-equal-to x; x is less-than 24.
    - But *only* use $<$ and $\leq$ for this (not $>$ or $\geq$).
    - Helps avoid confusion if we always write things smallest-to-largest.
- On the other hand, "0–23" works too.
  - *Provided* it's clear from the context whether 0 and 23 are *included* or *excluded* in the range!

Corner Cases

- There are some often-overlooked special values, particularly for strings and arrays.
  - Often not obvious what the expected result should be.
  - But, precisely because of this, they are important to test.
- Strings and arrays can be empty – zero elements long.
  - "" is a legitimate value.
  - What should `palindrome` return for this?
- Strings, arrays and other objects can be "`null`".
  - A special value indicating non-existence (*different* from being empty).
  - We'll tend to overlook this in ISE, for simplicity, but *in theory* it should be tested for.
- There may be other special cases too that depend on the situation.
  - e.g. we'd probably say 0 is a corner case for the `sgn` submodule.

Important Properties of Categories (1)

Categories must be complete.

- Every possible import value must fit into a category.
- e.g. For categories "$x < 5$" and "$x > 5$", where does 5 go?
- For submodules with multiple imports, every possible *combination* of import values must fit into a category.
  - e.g. `max` and `formatTime`.
- We want to cover all possibilities!

Important Properties of Categories (2)

Categories should not be joined with an "OR".

- *Don't* have a combined "$x < 0$ or $x \geq 100$" category.
- This effectively just removes test cases that may be important.
- The test data $x = -10$ (for instance) is *not* going to help test what happens when $x \geq 100$.

Important Properties of Categories (3)

Categories should not overlap.

- The whole point of identifying categories is to identify what individual test cases we need.
- They must be "mutually exclusive".
- A given set of import values cannot be in more than one category.
- e.g. "$x < 0$" and "$x < 100$" can't be two separate categories.
  - We probably meant "$x < 0$" and "$0 \le x < 100$".
- If the production code imports both x and y:
  - You can't have "$x < 0$" and "$y < 0$" as separate categories.
  - Each *pair* of values must fall into *one* category.
  - Which single category does $x = -10$, $y = -10$ fall into? What about $x = -5$, $y = 15$?

Important Properties of Categories (4)

Don’t try to test syntax errors.

- Remember sgn, which imports a real number:

> Submodule **sgn**
> Imports: **n (real)**
> Exports: **result (integer)**
> ...

- What if we pass sgn a string? Should we test that, and have a category for it?
- No! sgn *cannot* receive a string. The compiler prevents this.
- It’s not about error-handling. sgn doesn’t have to check it. It simply cannot happen.
- It’s silly trying to test something that doesn’t even compile!

Boundary Value Analysis (Another Black Box Technique)

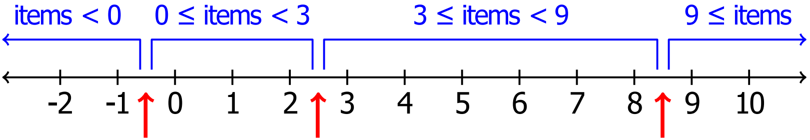
- BVA is a more intricate take on equivalence partitioning.
- Only applies to numerical imports (ranges of numbers).
- We look at the "boundaries" between categories.
    - A "boundary value" is one step away from being in a different category.
- Why? Largely because of "off-by-one" faults.
    - A common type of mistake made by coders, which may *only* show up on boundary values.
    - Writing $>=$ or $<=$ instead of $>$ or $<$, or vice versa.
    - Omitting "$-\ 1$" or a "$+\ 1$", or writing one where not needed.
    - Initialising a variable to 1 when it should be 0, or the other way around.

What are Boundaries?

- Consider a submodule dealing with ranges of integers: Submodule **discount**

  Imports: **items (integer)**
  Exports: **result (real)**
  Calculates a customer discount. For 0–2 items, there's no discount (0). For 3–8 items, a 15% discount (0.15). For 9 or more, 25%. For invalid amounts, return -1.0.

- The *boundaries* are where one category meets another:



Designing Test Cases with BVA

- Boundaries always lie *between two boundary values*.
- Those boundary values become the test data.
  - Implies there are *two* test cases for each boundary.
  - An off-by-one fault could go either way.
- Here's how we could lay it all out:

Boundary	Test Data	Expected Result
Invalid / 0%	items = −1 items = 0	−1.0 0.0
0 / 15%	items = 2 items = 3	0.0 0.15
15 / 25%	items = 8 items = 9	0.15 0.25

BVA With Real Numbers

- With real numbers, BVA requires some care.

> Submodule **tooHot**
> Imports: **temperature (real)**
> Exports: **aircon (boolean)**
> Determines whether it's hot enough to turn on the air conditioner. When the temperature is 25.5 degrees or higher, this returns true, and false otherwise.

- Our categories: temperature < 25.5, and temperature ≥ 25.5. Clearly 25.5 is the boundary.
- But what are the two *boundary values*?
- About 25.499999 (close enough), and also 25.5 exactly.
  - We need *one number from each category*.
  - 25.5 is on one side, so we need a value on the other.
  - It should be "as close to the edge" as possible.

Test Code

- Once designed, we implement our test cases in code.
- We write a "test suite".
  - This is a source file containing test methods/functions.
  - Each test method checks a specific production code method.
- To write a test method, we write code to do the following:
  1. Call the production code method, and pass it the test data.
  2. Receive the *actual* result from the production code.
  3. Compare the expected and actual results.
     - If they're equal, the test passes. Otherwise, it fails.
  4. Repeat, for each different test case.
     - Each test method will often run through several test cases.
- To do all this, we need to refer back to the test design.
  - We need (a) the test data, and (b) the expected results.

Test Code – Max Example

- Recall the test design for `max`?

Category	Test Data	Expected Result
value1 < value2	10, 20	20
value1 = value2	10, 10	10
value1 > value2	10, 5	10

- Take the first test case: value1 < value2.

- Here's how it would be implemented:

```java
int actual;
actual = MyUtils.max(10, 20); // Call prod. code.
assert 20 == actual;  // Compare expected to actual.
```

- `MyUtils.max(...)` is the call to the production code.
  - Assuming `max` is located inside `MyUtils.java`.
- "`10, 20`" is the test data; `20` is the expected result.

Test Code – Max Example

- Recall the test design for `max`?

Category	Test Data	Expected Result
value1 < value2	10, 20	20
value1 = value2	10, 10	10
value1 > value2	10, 5	10

- Take the first test case: value1 < value2.

Here's how it would be implemented:

```python
actual = MyUtils.max(10, 20) # Call prod. code.
assert 20 == actual     # Compare expected to actual.
```

- `MyUtils.max(...)` is the call to the production code.
  - Assuming `max` is located inside `MyUtils.java`.
- "`10, 20`" is the test data; `20` is the expected result.

Assertion Statements

- In Java, an assertion statement looks like either of these:
  - `assert condition;`
  - `assert condition : "message";`
- Very similar in Python:
  - `assert condition`
  - `assert condition, "message"`
- The condition is a *boolean expression* – something that is either true or false; e.g.
  - `assert apples == bananas;` (`apples` is equal to `bananas`.)
- In practice, we're almost always checking for equality.
- If the condition is true, nothing happens.
- If the condition is false, the test aborts, displaying the message (if any).

Checking Equality (Java)

- To compare integers, characters or booleans, use "==":
  - `assert x == y;`
- To compare strings (and other objects), use the "`equals`" method; e.g.
  - `assert x.equals(y);`
- For booleans, we can also just assert the value directly:
  - `assert x;` (`x` should contain the value '`true`')
  - `assert !y;` (`y` should contain the value '`false`')
- To compare real numbers, use a tolerance value.
  - If the difference between x and y is less than a very small number (the tolerance), we consider x and y equal.
  - `assert Math.abs(x - y) < 0.000001;`
- *Don't* use "`=`" inside assertions. This *assigns* rather than compares values.

Checking Equality (Python)

- To compare *everything except real numbers* in Python, use "==":
  - `assert x == y`
- For booleans, we can also just assert the value directly:
  - `assert x` (x should contain the value 'True')
  - `assert not y` (y should contain the value 'False')
- To compare real numbers, use a tolerance value.
  - If the difference between x and y is less than a very small number (the tolerance), we consider x and y equal.
  - `assert abs(x - y) < 0.000001`

FYI: Assertions in Production Code

- Assertions can appear anywhere in your algorithm.
- Useful as a “sanity check” on your code.
- Just don’t misuse them!
  - Never use assertions to actually *do* something in your algorithm.
  - Never use assertions for validating user input. (In most cases, your program shouldn’t abort just because the user entered the wrong number.)
  - Assertions should never fail unless your program is faulty.
    - If they do, you’re misusing assertions.
- Assertions bring faults to your attention, so you can find and fix them.
  - They won’t catch every fault, though!

Assertion Messages

- Add messages to your assertions:

Java `assert condition : "message";`

Python `assert condition, "message"`

- Messages should be written carefully, to provide more information on what is being tested.
- When a test fails, the message should help you understand *which* test failed.
  - You could find this out anyway, but the message should help you find it *quicker*.
- To this end, you could embed the test import(s) in the message string.

Max Test Code

- Here's the implementation of all three `max` test cases:

Java

```
public static void testMax()
{
    int actual;
    actual = MyUtils.max(10, 20);
    assert 20 == actual : "value1 < value2";

    actual = MyUtils.max(10, 10);
    assert 10 == actual : "value1 = value2";

    actual = MyUtils.max(10, 5);
    assert 10 == actual : "value1 > value2";
}
```

- This *isn't* the complete test file – we'll get to that.

Max Test Code

- Here's the implementation of all three `max` test cases:

Python

```
def testMax():
    actual = MyUtils.max(10, 20)
    assert 20 == actual, "value1 < value2"

    actual = MyUtils.max(10, 10)
    assert 10 == actual, "value1 = value2"

    actual = MyUtils.max(10, 5)
    assert 10 == actual, "value1 > value2"
```

- This *isn't* the complete test file – we'll get to that.

Palindrome Test Code

- Recall the `palindrome` test design?

Category	Test Data	Expected Result
s *is* a palindrome	"glenelg"	true
s *isn't* a palindrome	"albuquerque"	false

- Here's how we'd implement that:

Java

```
public static void testPalindrome()
{
    boolean actual;
    actual = MyUtils.palindrome("glenelg");
    assert actual;  // OR assert actual == true;
    actual = MyUtils.palindrome("albuquerque");
    assert !actual; // OR assert actual == false;
}
```

Palindrome Test Code

- Recall the `palindrome` test design?

Category	Test Data	Expected Result
s *is* a palindrome	"glenelg"	true
s *isn't* a palindrome	"albuquerque"	false

- Here's how we'd implement that:

Python

```
def testPalindrome():
    actual = MyUtils.palindrome("glenelg")
    assert actual      # OR assert actual == True

    actual = MyUtils.palindrome("albuquerque")
    assert not actual # OR assert actual == False
}
```

FormatTime Test Code

And (some of) the `formatTime` test design and implementation:

Category	Test data	Expected Result
inHours, inMins	inHours, inMins	
0–23, 0–59	12, 30	"12:30"
0–23, < 0	12, -10	"error"
...	...	...

Java

```java
public static void testFormatTime()
{
    String actual;
    actual = MyUtils.formatTime(12, 30);
    assert "12:30".equals(actual) : "valid";
    actual = MyUtils.formatTime(12, -10);
    assert "error".equals(actual) : "mins negative";
    ...
```

FormatTime Test Code

And (some of) the `formatTime` test design and implementation:

Category inHours, inMins	Test data inHours, inMins	Expected Result
0–23, 0–59	12, 30	"12:30"
0–23, < 0	12, -10	"error"
...	...	...

Python

```python
def testFormatTime():
    actual = MyUtils.formatTime(12, 30)
    assert "12:30" == actual, "valid"

    actual = MyUtils.formatTime(12, -10)
    assert "error" == actual, "mins negative"
    ...
```

Too Verbose?

- We don't actually need to *store* "`actual`". We can just use the return value directly:

```java
public static void testMax()
{
    assert 20 == MyUtils.max(10, 20) : "value1 < value2";
    assert 10 == MyUtils.max(10, 5) : "value1 = value2";
    assert 10 == MyUtils.max(10, 10) : "value1 > value2";
}
```

```java
public static void testFormatTime()
{
    assert "12:30".equals(MyUtils.formatTime(12, 30)): "...";
    assert "error".equals(MyUtils.formatTime(12, -10)): "...";
    ...
```

Too Verbose?

- We don't actually need to *store* "`actual`". We can just use the return value directly:

```python
def testMax():
    assert 20 == MyUtils.max(10, 20), "value1 < value2"
    assert 10 == MyUtils.max(10, 5), "value1 = value2"
    assert 10 == MyUtils.max(10, 10), "value1 > value2"
```

```python
def testFormatTime():
    assert "12:30" == MyUtils.formatTime(12, 30), "..."
    assert "error" == MyUtils.formatTime(12, -10), "..."
    ...
```

Arrays/Lists and For Loops

- We can also put the test data and expected results (and messages) in arrays.
- We can then put a single `assert` in a `for` loop.
- Makes sense if we have *lots* of test cases.

Arrays/Lists and For Loops (Java)

```
public static void testMax()
{
    int[] x = {10, 10, 10};           // This isn't "lots" of
    int[] y = {20, 10, 5};            // test cases, but just
    int[] exp = {20, 10, 10};         // for illustration.
    String[] msg = {"x < y", "x = y", "x > y"};

    for(int i = 0; i < x.length; i++)
    {
        assert exp[i] == MyUtils.max(x[i], y[i]) : msg[i];
    }
}
```

Arrays/Lists and For Loops (Python)

```
def testMax():
    x = [10, 10, 10]                    # This isn't "lots" of
    y = [20, 10, 5]                     # test cases, but just
    exp = [20, 10, 10]                  # for illustration.
    msg = ["x < y", "x = y", "x > y"]

    for i in range(len(x)):
        assert exp[i] == MyUtils.max(x[i], y[i]), msg[i]
```

Test Suites: Putting it Together (Java)

- We need a proper class structure around our test code too:

```
public class MyUtilsTest    // Save to 'MyUtilsTest.java'
{
    public static void main(String[] args)
    {
        testMax();
        testPalindrome();
        testFormatTime();
    }
    public static void testMax() { ... }
    public static void testPalindrome() { ... }
    public static void testFormatTime() { ... }
}
```

- This is a complete test suite (where ". . . " is the test code).
- `main()` simply calls each of our test methods in turn.

Test Suites: Putting it Together (Python)

- We need to import the production code, and call the test functions:

```
import MyUtils                    # Save to 'testMyUtils.py'

def testMax(): ...
def testPalindrome(): ...
def testFormatTime(): ...

if __name__ == "__main__":
    testMax()
    testPalindrome()
    testFormatTime()
```

- This is a complete test suite (where "..." is the test code).
- The last section simply calls each of our test functions in turn.

Running Tests (Java)

- Finally, we're ready to compile and run the test code:

```
[user@pc]$ javac MyUtilsTest.java
```

```
[user@pc]$ java -ea MyUtilsTest
```

- "`-ea`" means "enable assertions".
  - By default, `assert` doesn't actually do anything!
  - We must enable assertions, or our tests will pass even when they should fail (which is very bad!)
- If all tests pass, nothing happens. If one fails, you'll see this:

```
Exception in thread "main" java.lang.AssertionError: x < y
        at MyUtilsTest.testMax(MyUtilsTest.java:15)
        at MyUtilsTest.main(MyUtilsTest.java:5)
```

Running Tests (Python)

- Finally, we're ready to compile and run the test code:

```
[user@pc]$ python testMyUtils.py
```

- If all tests pass, nothing happens. If one fails, you'll see this:

```
Traceback (most recent call last):
File "testMyUtils.py", line 21, in <module>
    testMax()
File "testMyUtils.py",        , in testMax

AssertionError: x < y
```

Unit Testing Frameworks

- What we just saw was pure Java/Python.
- In practice, unit testing mostly uses a “unit test framework”.
- Several advantages:
  - All your tests will be run, even if one fails.
    - (The way we did things before, the whole test suite ends if/when a single test fails.)
  - More meaningful error messages.
    - When things fail, a good informative error message can save you a lot of work.
  - For Java, no need to remember to “enable assertions”.
- Different languages have different unit test frameworks.

  Java: the third-party “JUnit” library.

  Python: the standard “`unittest`” module.

JUnit (Java)

JUnit makes a few key differences to how we write test code:

1. Delete the `main` method – we don't need it.
2. Delete the `static` keyword from each test method.
3. Put "`@Test`" in front of each test method, and "`@RunWith(JUnit4.class)`" in front of the test suite class.
   - We're basically telling JUnit what tests it has to run.
4. Replace `assert` with JUnit's enhanced assertions.
   - In particular, `assertEquals(...)`, and `assertTrue(...)`.
5. Place these lines at the top of your test code:
   - Tells the compiler about JUnit.
   - Just copy and paste. You don't have to memorise this!

```
import org.junit.*;
import  org.junit.runner.RunWith;
import  org.junit.runners.JUnit4;
import static org.junit.Assert.*;
```

JUnit Test Suite Example (Java)

Here's our test suite from before, rewritten for JUnit:

```java
import org.junit.*; ... // And other import statements...

@RunWith(JUnit4.class)
public class MyUtilsTest
{
    @Test
    public void testMax() { ... }

    @Test
    public void testPalindrome() { ... }

    @Test
    public void testFormatTime() { ... }
}
```

JUnit Test Method Example (Java)

```
public static void testMax()    // Non-JUnit (for comparison)
{
    assert 20 == MyUtils.max(10, 20) : "value1 < value2";
    assert 10 == MyUtils.max(10, 5)  : "value1 = value2";
    assert 10 == MyUtils.max(10, 10) : "value1 > value2";
}
```

```
@Test                                              // With JUnit
public void testMax()
{
    assertEquals("value1 < value2", 20, MyUtils.max(10, 20));
    assertEquals("value1 = value2", 10, MyUtils.max(10, 5));
    assertEquals("value1 > value2", 10, MyUtils.max(10, 10));
}
```

Enhanced Assertions (Java)

- JUnit provides alternatives to the standard Java `assert` statement.
  - (These are technically methods, not language constructs.)

`assertEquals(message, expected, actual);`
: Checks that `expected` and `actual` are equal. (These can be integers, strings or other objects.)

`assertEquals(message, expected, actual, delta);`
: Checks that real numbers `expected` and `actual` are equal, ignoring rounding errors (i.e. within `delta` of each other, where `delta` should be something like 0.0001).

`assertTrue(message, x);` – Checks that boolean value `x` is true.

`assertFalse(message, x);` – Checks that `x` is false.

- . . . and others.
- The `message` is optional.

`assertEquals` Arguments (Java)

- `assert` requires a boolean condition (true/false).
- `assertEquals` does not. It takes the expected and actual results directly:

$$\texttt{assertEquals(}\underbrace{\texttt{"x < y"}}_{\text{message}}\texttt{, }\overbrace{\texttt{20}}^{\text{expected}}\texttt{, }\underbrace{\texttt{MyUtils.max(10, 20)}}_{\text{actual}}\texttt{);}$$

- i.e. Don't write "`x == y`" or "`x.equals(y)`" here.
- For real numbers, you need a tolerance ("delta"):

```
assertEquals("message", 5.0, actual, 0.0001);
```

Running JUnit: The Basic Command-Line Version (Java)

- JUnit is almost always run via an IDE or a "build tool".
- But here's the basic command-line procedure, just in case.
- Step 1: locate the file "`junit.jar`".
  - e.g. it might be "`/usr/share/junit-4/lib/junit.jar`", or "`C:\Users\Myself\Desktop\junit.jar`", or somewhere else[1].
  - I'm just going to write "**junit-path**" to represent this.
- Step 2: compile.

```
$ javac -cp junit-path MyUtilTest.java
```

- Step 3: run JUnit.

```
$ java -cp junit-path org.junit.runner.JUnitCore MyUtilTest
```

- (Replace `MyUtilTest` with your test suite's name.)

[1] If necessary, download JUnit from https://search.maven.org/remotecontent?filepath=junit/junit/4.12/junit-4.12.jar

Running JUnit: With ise-test.zip (Java)

- For ISE purposes, you can instead download ise-test.zip (from Blackboard), and use it to run JUnit tests.
- It contains `unittest`, `unittest.bat`, and `buildsystem` (a directory).
- Put these in the same directory as your .java files.
- Then simply run the script.
  - On Linux/macOS:

```
[user@pc]$ ./unittest
```

  - On Windows:

```
C:\Users\Myself\ISE\> unittest
```

- Behind the scenes, this will download and run a build tool called "Gradle".
- Gradle will compile and run all JUnit tests (within the directory) automatically.

`unittest` (Python)

`unittest` also makes some differences to how we write test code:

1. Delete the "`if __name__ == "__main__"`" section.
2. Put all the test methods into a "`class`" declaration:

```
class TestSuite(unittest.TestCase):
    ... # Your test methods here
```

   - You don't really have to understand what classes are.
   - This basically just lets `unittest` find and run your test code.
3. Give each test method a "`self`" parameter.
   - This is part of how "classes" work in Python.
4. Replace `assert` with `unittest`'s enhanced assertions.
   - In particular, `assertEqual()`, `assertAlmostEqual()`, `assertTrue()` and `assertFalse()`.
5. Place this line at the top of your test code:

```
import unittest
```

unittest Test Suite Example (Python)

Here's our test suite from before, rewritten for unittest:

```
import MyUtils      # Our production code
import unittest     # The test framework

class MyUtilsTest(unittest.TestCase):
    def testMax(self): ...
    def testPalindrome(self): ...
    def testFormatTime(self): ...
```

- unittest will automatically find and run all methods starting with "test".

unittest Test Method Example (Python)

```
def testMax():          # Non-'unittest' test (for comparison)
    assert 20 == MyUtils.max(10, 20), "v1 < v2";
    assert 10 == MyUtils.max(10, 5), "v1 = v2";
    assert 10 == MyUtils.max(10, 10), "v1 > v2";
```

```
def testMax(self):      # With unittest
    self.assertEqual(20, MyUtils.max(10, 20), "v1 < v2");
    self.assertEqual(10, MyUtils.max(10, 5), "v1 = v2");
    self.assertEqual(10, MyUtils.max(10, 10), "v1 > v2");
```

Enhanced Assertions (Python)

`unittest` provides alternatives to the standard `assert` statement.

```
self.assertEqual(<expected>, <actual>, <message>)
```

- Checks that expected and actual are equal.

```
self.assertAlmostEqual(<expected>, <actual>,
    delta = <delta>, msg = <message>)
```

- Checks that expected and actual (real values) are very close; i.e. the difference is less than delta.

```
self.assertTrue(<x>, <message>)
self.assertFalse(<x>, <message>)
```

- Checks that boolean value x is true (or false).

assertEqual Arguments (Python)

- assertEqual takes the expected and actual results directly:

```
self.assertEqual( 20, MyUtils.max(10, 20), "x < y");
```

$\underbrace{\texttt{20}}_{\text{expected}}$, $\underbrace{\texttt{MyUtils.max(10, 20)}}_{\text{actual}}$, $\underbrace{\texttt{"x < y"}}_{\text{message}}$

- i.e. Don't write "x == y".
- For real numbers, you must call assertAlmostEqual with a tolerance ("delta"):

```
self.assertAlmostEqual(5.0, actual, delta = 0.0001,
                       msg = "message");
```

Running `unittest` (Python)

- We can simply run `unittest` like this:

```
[user@pc]$ python -m unittest
```

- It will look for all `test*.py` files in the current directory (starting with "test", ending in ".py").
- It will run any test suites in them, and display the results.
- (Note: `ise-test.zip` is only for Java, not Python.)

That's all for now!