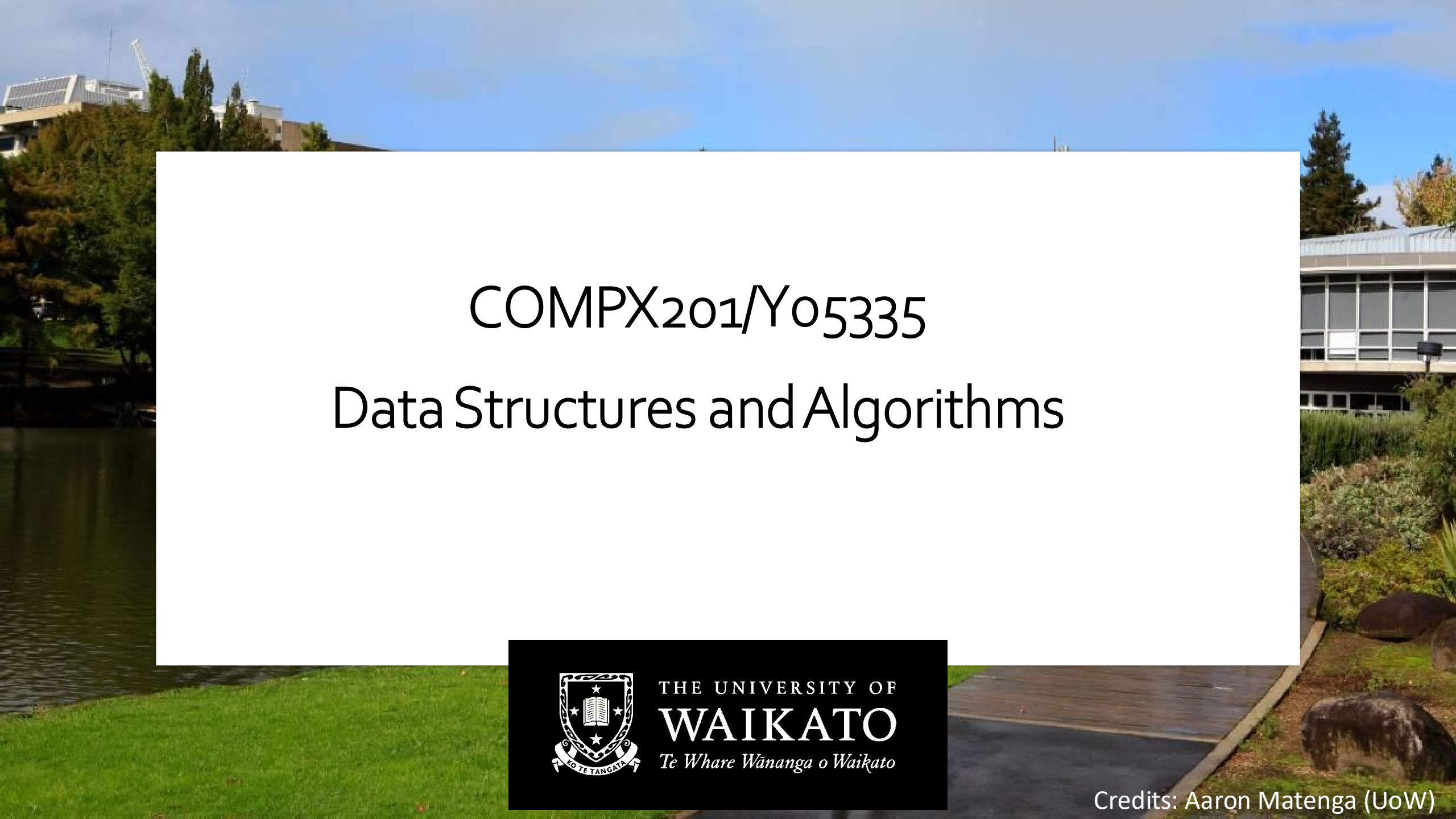


COMPX201/Y05335

Data Structures and Algorithms

THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato

Credits: Aaron Matenga (UoW)

Unit Testing
An Introduction

COMPX201/Y05335

Overview

- Reminder: what are unit tests?
- JUnit 5
- How to write tests with JUnit

Reminder: Unit testing

Unit testing

- Tests individual *units* of source code
- Typically automated

- A unit can be:
  - A module
  - A class
  - A method

- Parametrized unit tests
  - Run the same test multiple times with different parameters

Unit testing

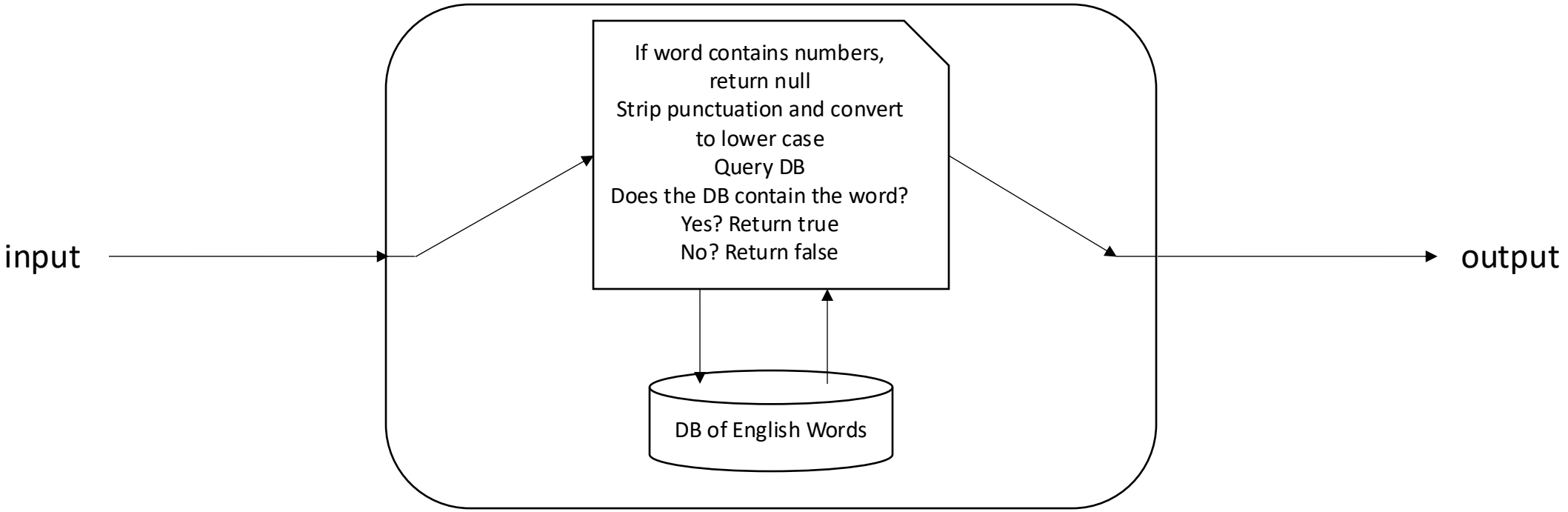
- Each test case should be tested independently
- This allows us to isolate each potential issue

- Can use unit tests to write tests for the smallest testable units, then combine tests to build up a comprehensive set of tests for complex applications

Unit testing example

"We would like to create a spell checker application. The application will take a word and either confirm that it is spelt correctly or confirm that it is not spelt correctly. Input words may include punctuation, but cannot include numerical characters"

Unit testing example



Unit testing example

- Units:
  - Write a unit test to test the method that disregards strings with numerical values
  - Write a unit test to test the method that strips punctuation
  - Write a unit test to test the method that converts to lower case

JUnit 5

JUnit 5

- A very common package for writing, organising, and running tests in Java
- You will need to set up your environment for JUnit

Writing tests with JUnit

A JUnit Test method has 3 requirements:

1. It isn't static or private
2. It can't return anything (must be void)
3. A test method must be annotated with the @Test notation

Writing tests with JUnit

In JUnit, annotations are used to specify when methods should run during testing.

Some common annoations include:

- @BeforeAll: execute this method before all tests
- @BeforeEach: execute this method before each test
- @AfterAll: execute this method after all tests
- @AfterEach: execute this method after each test
- @Test: the following method is a test
- @DisplayName: the displayed name of the test method

Writing tests with JUnit

In JUnit, values are evaluated using "assertions".

Some common assertions include:

- **assertEquals(x, y):** asserts that expected (x) and actual (y) are equal
- **assertNotEquals(x, y):** asserts that expected (x) and actual (y) are not equal
- **assertTrue(boolean condition):** asserts that the supplied condition is true
- **assertFalse(boolean condition):** asserts that the supplied condition is not true
- **assertNotNull(Object actual):** asserts that actual is not null
- … and lots more!

https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html

For example …

Assume that we have the following class:

```
public class HelloWorld {

    public String helloWorld(){
        return "Hello World!";
    }

}
```

We want to test the “helloWorld()” method to determine if it returns the correct string.

Example Test

```
@Test
@DisplayName("Test Hello World Method")
void helloWorldTest() {
    HelloWorld t = new HelloWorld();
    Assertions.assertEquals(t.helloWorld(), "Hello World!");
}
```

Note

To use JUnit, we need to:

1. Create a Test class
2. Import JUnit
3. Add Test methods
4. Compile and run our classes using JUnit

Example Test

```
import org.junit.jupiter.api.*;
import org.junit.jupiter.api.Assertions.*;

class ExampleTest {

    @Test
    @DisplayName("Test Hello World Method")
    void helloWorldTest() {
        HelloWorld t = new HelloWorld();
        Assertions.assertEquals(t.helloWorld(), "Hello World!");
    }
}
```

Compile and run using JUnit

First, we need to download the *junit-platform-console-standalone-1.8.2.jar* file from the JUnit 5 website, and store it in the same directory as our Java files.

https://junit.org/junit5/docs/current/user-guide/#running-tests-console-launcher

Compile and run using JUnit

Next, assuming all files are in the same directory, enter the following command to compile your Java files:

```
javac -cp "junit-platform-console-standalone-1.8.2.jar" *.java
```

This will create the correct .class files so that you may run the tests.

Example Test

Finally, in the same directory, run the test with the following command:

```
java -jar junit-platform-console-standalone-1.8.2.jar -cp .\ -c ExampleTest
```

This tells java to run the standalone version of JUnit and to look for the test “ExampleTest”.

Example Test

- Note:
  - Windows: `java -jar junit-platform-console-standalone-1.8.2.jar -cp .\ -c ExampleTest`
  - Linux/Mac: `java -jar junit-platform-console-standalone-1.8.2.jar -cp "." -c ExampleTest`

Example Test

If your test ran successfully you should get something similar to the following output:

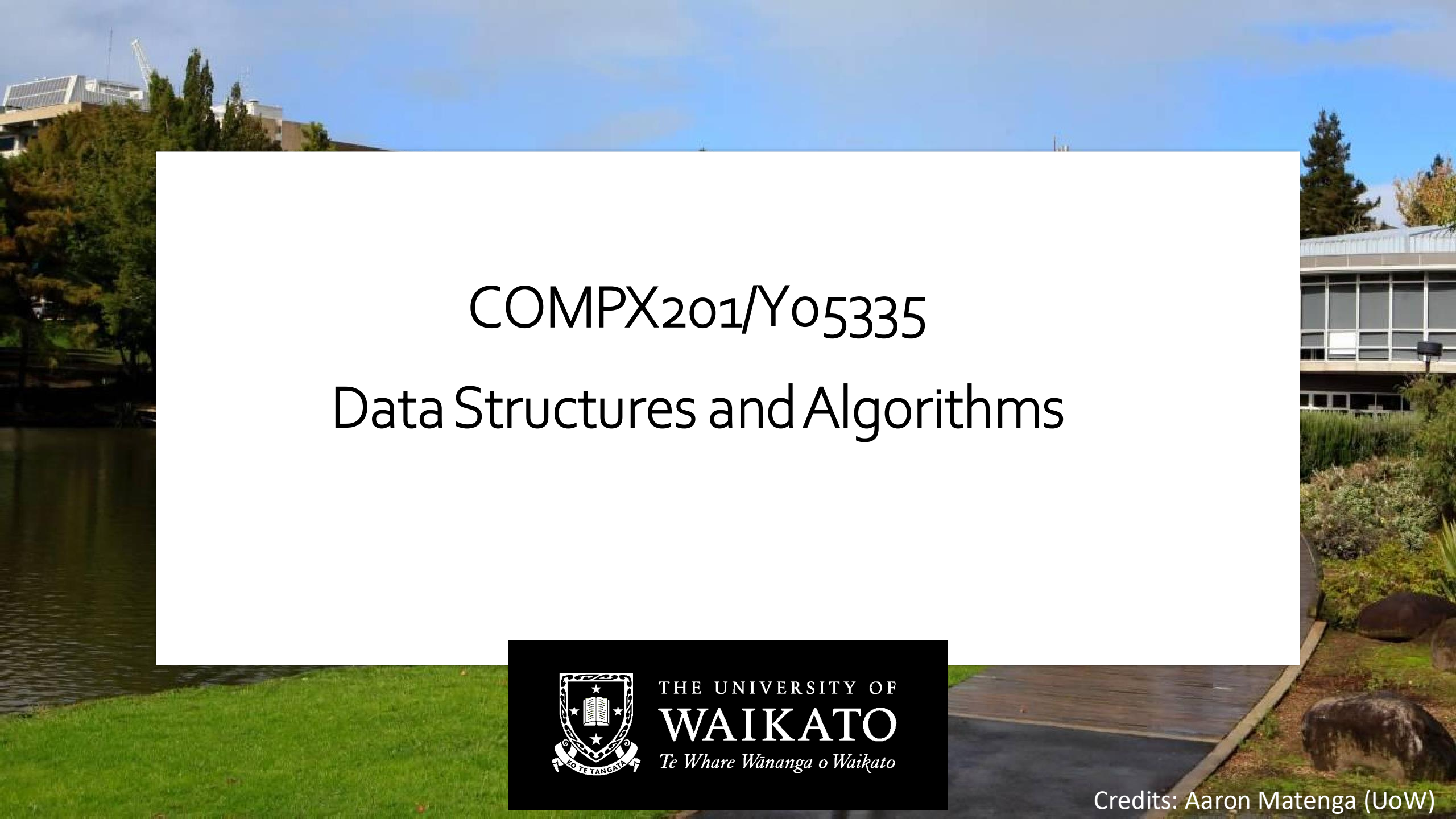
```
Thanks for using JUnit! Support its development at https://junit.org/sponsoring

╷
├─ JUnit Jupiter ✓
│  └─ ExampleTest ✓
│     └─ Test Hello World Method ✓
└─ JUnit Vintage ✓

Test run finished after 64 ms
[         3 containers found      ]
[         0 containers skipped    ]
[         3 containers started    ]
[         0 containers aborted    ]
[         3 containers successful ]
[         0 containers failed     ]
[         1 tests found           ]
              ...
```

Example Test

- Okay, lets try it …



COMPX201/Y05335

Data Structures and Algorithms

THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato

Credits: Aaron Matenga (UoW)

Unit Testing
Designing Tests

COMPX201/Y05335

Overview

- Designing tests
  - How to test
  - What to test
- Example

https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices
https://betterprogramming.pub/13-tips-for-writing-useful-unit-tests-ca20706b5368

Designing tests

How to test:

1. Test in isolation
2. Write minimally passing tests
3. Follow the AAA (Arrange, Act, Assert) rule
4. Avoid multiple Acts

What to test:

1. Test expected cases
2. Test edge cases
3. Test across boundaries
4. Test the entire spectrum (only if you can!)
5. Test exceptions

How to test:

1. Test in isolation
2. Write minimally passing tests
3. Follow the AAA rule
4. Avoid multiple Acts

Test in isolation

- Unit tests should be standalone
- Should be run in isolation
- Should have no dependencies on any outside factors such as another Class, a file system or database.

Write minimally passing tests

- The input should be the simplest possible
- “Tests that include more information than required to pass the test have a higher chance of introducing errors into the test and can make the intent of the test less clear. ” https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices

- For example:

```
Assertions.assertEquals(4237, stringCalculator.Add("4237"));
```

versus

```
Assertions.assertEquals(1, stringCalculator.Add(“1"));
```

Follow the AAA rule

- Each test should follow the *Arrange*, *Act*, *Assert* structure
  1. Arrange your objects, creating and setting them up as necessary
  2. Act on an object
  3. Assert that something is as expected

- For example:

```
// Arrange
StringCalculator stringCalculator = new StringCalculator();
// Act
int actual = stringCalculator.Add(“1");
// Assert
Assertions.assertEquals(1, stringCalculator.Add(“1"));
```

Avoid multiple Acts

- Don’t test multiple cases (*Acts*) in the same Unit Test

- For example, do not do this:

```
StringCalculator stringCalculator = new StringCalculator();
int actual1 = stringCalculator.Add(“1");
int actual2 = stringCalculator.Add(“-1");
Assertions.assertEquals(1, actual1);
Assertions.assertEquals(-1, actual2);
```

- If the test fails, you won’t know which condition failed.
- Instead, create seperate unit tests
- Or use parametized unit tests

What to test:

1. Test expected cases
2. Test edge cases
3. Test across boundaries
4. Test the entire spectrum (only if you can!)
5. Test exceptions

Test expected cases

- Write obvious tests first
- i.e. tests that move through the main path of your code
- Once these tests are written, you can move on to edge and boundary cases

- E.g. If testing an addition function, start by testing that 2 + 2 = 4

Test edge cases

- An edge case is one that tests the outer limits of a method
- E.g. If testing an addition function, you may create a test using very small numbers, and a test that uses very large numbers.
- You can then assume that, if it works for both ends of the spectrum, it will work for inbetween.

Test boundary cases

- Boundary tests should test both sides of a boundary.
- E.g. If testing an addition function, you may create a test using a negative number, and a test that uses a positive number.
- E.g. If testing a date/time function, you may test one second before midnight, or one second after midnight

Test the entire spectrum

- There may be some cases where you can test all possible inputs – if you can, then you should
- E.g. If testing an addition function, you cannot and should not test all possible integers!
- But, if testing a function that has a smaller finite set of possible inputs, then you could and should test the full spectrum

Test exceptions

- If there are places in your code where you raise an exception, then you should test that these exceptions are raised when expected

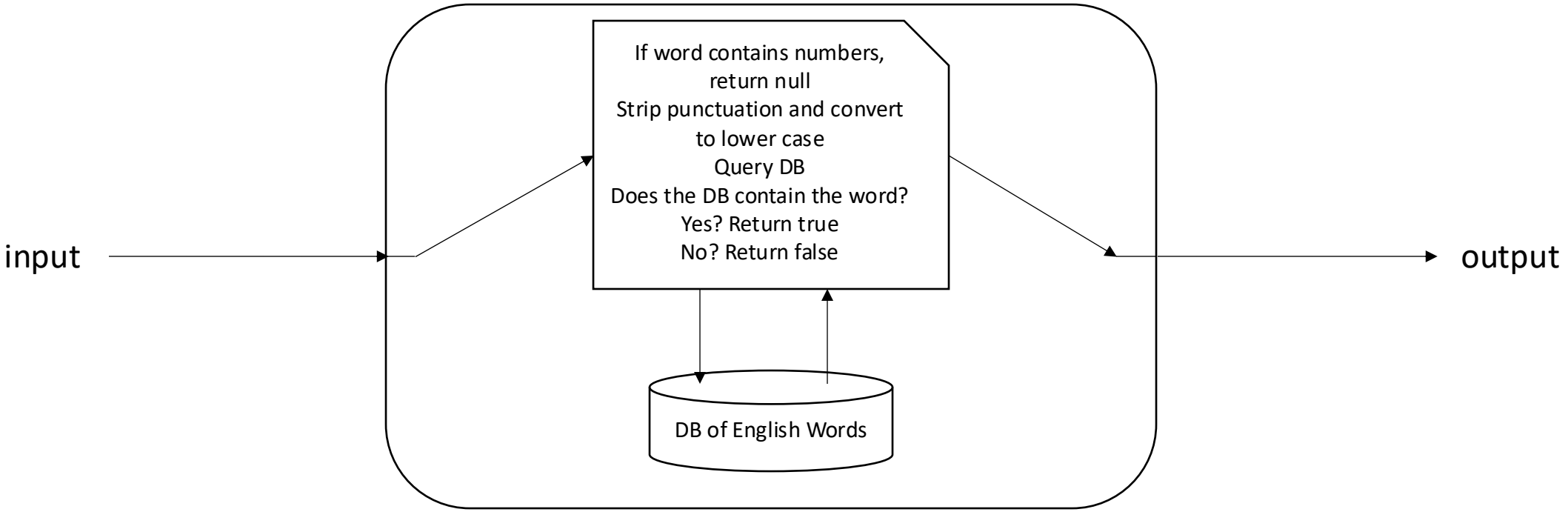
- E.g. If testing an addition function and you raise an exception if the wrong datatype is passed through, then test that the exception is raised when expected

Unit testing example

Unit testing example

"We would like to create a spell checker application. The application will take a word and either confirm that it is spelt correctly or confirm that it is not spelt correctly. Input words may include punctuation, but cannot include numerical characters"

Unit testing example



Unit testing example

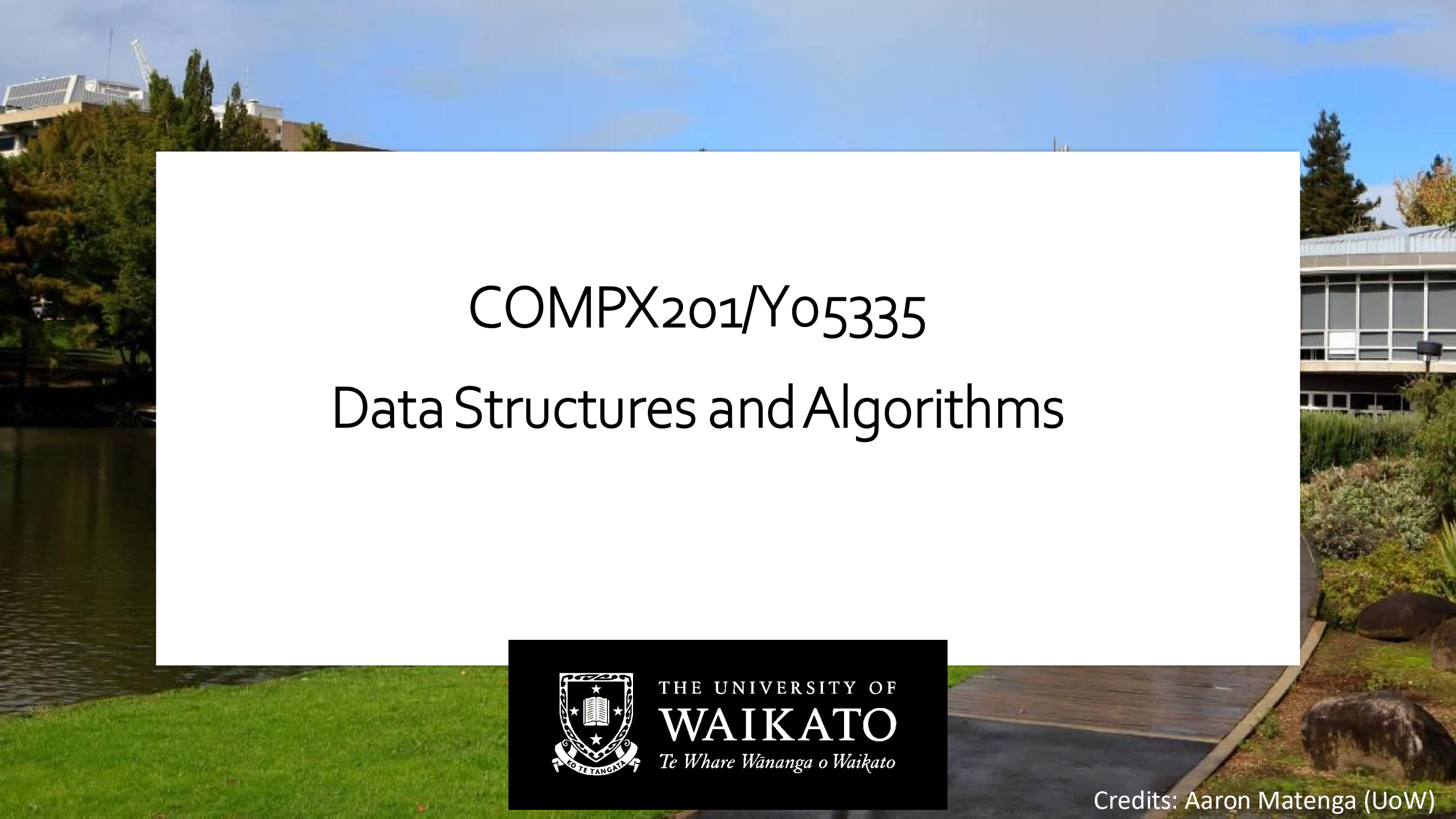
- Units:
  - Write a unit test to test the method that disregards strings with numerical values
  - Write a unit test to test the method that strips punctuation
  - Write a unit test to test the method that converts to lower case

Unit testing example

- Units:
  - Write a unit test to test the method that disregards strings with numerical values
  - Write a unit test to test the method that strips punctuation
  - Write a unit test to test the method that converts to lower case

Unit testing example

- Units:
  - Write a unit test to test the method that disregards strings with numerical values
    - Input? Expected output?
      - a string without numerical values
      - a string with numerical values
      - an empty string
  - Write a unit test to test the method that strips punctuation
  - Write a unit test to test the method that converts to lower case



COMPX201/Y05335

Data Structures and Algorithms

THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato

Credits: Aaron Matenga (UoW)