



Welcome to this CoGrammar lecture: Unit Testing

The session will start shortly...

Questions? Drop them in the chat.
We'll have dedicated moderators
answering questions.



Software Engineering Session Housekeeping

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly.
(Fundamental British Values: Mutual Respect and Tolerance)
- No question is daft or silly - **ask them!**
- There are **Q&A sessions** throughout this session, should you wish to ask any follow-up questions.
- If you have any questions outside of this lecture, or that are not answered during this lecture, please do submit these for upcoming Academic Sessions. You can submit these questions here: [Questions](#)

Software Engineering Session Housekeeping cont.

- For all **non-academic questions**, please submit a query: www.hyperiondev.com/support
- Report a **safeguarding** incident: www.hyperiondev.com/safeguardreporting
- We would love your **feedback** on lectures: [Feedback on Lectures](#)
- If you are hearing impaired, please kindly use your computer's function through Google chrome to enable captions.

Safeguarding & Welfare

We are committed to all our students and staff feeling safe and happy; we want to make sure there is always someone you can turn to if you are worried about anything.

If you are feeling upset or unsafe, are worried about a friend, student or family member, or you feel like something isn't right, speak to our safeguarding team:



Ian Wyles
Designated Safeguarding
Lead



Simone Botes



Nurhaan Snyman



Rafiq Manan



Ronald Munodawafa



Tevin Pitts

Scan to report a
safeguarding concern



or email the Designated
Safeguarding Lead:
Ian Wyles

safeguarding@hyperiondev.com

A background image showing three people in a classroom or office setting. Two people are standing and looking at a laptop screen, while a third person is sitting and looking at the same screen. The image is dark and has a green tint.

CoGrammar

Unit Testing

**SKILLS
FOR LIFE**

SKILLS BOOTCAMPS



Department
for Education

Learning Outcomes

- Define unit testing and explain its use in software engineering.
- Use arrange, act, assert pattern for structuring tests.
- Describe the FIRST principles and how following these principles leads to clear, fast and accurate tests.
- Implement unit testing within your projects to test behaviour.
- Refactor code to resolve failing tests.

Unit Testing



What is unit testing?

- Software testing method where individual units or components of a software application are tested in isolation to ensure they work as intended.
- The goal is to verify that each unit of the software performs as designed and that all components are working together correctly.
- Unit tests help developers catch bugs early in the development process, when they are easier and less expensive to fix.
- It also helps ensure that any changes made to the code do not cause unintended consequences or break existing functionality.

Advantages of Unit Testing

- Catch errors early
- Improve code quality
- Refactor with confidence
- Document code behaviour
- Facilitate collaboration

Arrange, Act, Assert



Arrange, Act, Assert

- The AAA pattern is a common pattern used in unit testing to structure test cases. It stands for Arrange, Act, Assert.
 - **Arrange**: Set up any necessary preconditions or test data for the unit being tested.
 - **Act**: Invoke the method or code being tested.
 - **Assert**: Verify that the expected behaviour occurred.
- Using the AAA pattern helps make unit tests more **readable** and **easier to maintain**. It also helps **ensure** that **all** necessary **steps** are taken to **properly test** the unit being tested.

Arrange, Act, Assert

- Let's have a look at an example of how to write a unit test in Python using the AAA pattern.
- Consider a simple function that adds two numbers:

```
def add_numbers(a, b):  
    return a + b
```

Arrange, Act, Assert

- To test this function, we would create a new function called `test_add_numbers` (note that the name must start with `test_` for the Python test runner to find it).

```
def test_add_numbers(self):  
    # Arrange  
    a = 2  
    b = 3  
  
    # Act  
    result = add_numbers(a, b)  
  
    # Assert  
    self.assertEqual(result, 5)
```

We've set up the test data (**Arrange**) by creating two variables **a** and **b** with the values 2 and 3.

We then invoke the function being tested (**Act**) and store the result in a variable called **result**.

Finally, we assert that the result is equal to the expected value of **5** (**Assert**).

Assert Methods List

Method	Checks that	New in
<code>assertEqual(a, b)</code>	<code>a == b</code>	
<code>assertNotEqual(a, b)</code>	<code>a != b</code>	
<code>assertTrue(x)</code>	<code>bool(x) is True</code>	
<code>assertFalse(x)</code>	<code>bool(x) is False</code>	
<code>assertIs(a, b)</code>	<code>a is b</code>	3.1
<code>assertIsNot(a, b)</code>	<code>a is not b</code>	3.1
<code>assertIsNone(x)</code>	<code>x is None</code>	3.1
<code>assertIsNotNone(x)</code>	<code>x is not None</code>	3.1
<code>assertIn(a, b)</code>	<code>a in b</code>	3.1
<code>assertNotIn(a, b)</code>	<code>a not in b</code>	3.1
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>	3.2
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>	3.2

Source:
<https://docs.python.org/3/library/unittest.html#assert-methods>

FIRST



FIRST

- Set of rules created by uncle bob also known for the SOLID principles and TDD.
- We follow these rules when creating tests to make sure our tests are clear, simple and accurate.
- FIRST – **F**ast, **I**ndependent, **R**epeatable, **S**elf Validating and **T**horough

FIRST

- Fast
 - Tests should be fast and can run at any point during the development cycle.
 - Even if there are thousands of unit tests it should run and show the desired outcome it seconds.
- Independent
 - Each unit test, its environment variables and setup should be independent of everything else.
 - Our results should not be influenced by other factors.
 - Should follow the 3 A's of testing: Arrange, Act, Assert

FIRST

- Repeatable
 - Tests should be repeatable and deterministic, their values shouldn't change based on being run on different environments.
 - Each test should work with its own data and should not depend on any external factors to run its test
- Self Validating
 - You shouldn't need to check manually, whether the test passed or not.

FIRST

- Thorough
 - Try covering all the edge cases.
 - Test for illegal arguments and variables.
 - Test for security and other issues
 - Test for large values, what would a large input do.
 - Should try to cover every use case scenario and not just aim for 100% code coverage.

unittest



CoGrammar

Unit Tests

- Different packages for unit testing - Pytest, unittest, testify, Robot
- We will use **unittest**. It is built into python and does not require additional installations.
- To use unittest we simply import the module and create a class for our testing.

```
import unittest  
  
class TestExamples(unittest.TestCase)
```

Unit Tests

- Let's take a look at some behaviour we can test using unittest.
- Note that a unit does not necessarily mean a function but refers to behaviour within our program.
- Some units under test will use more than one function for its intended behaviour.

```
def sum_list(num_list):  
    total = 0  
    for num in num_list:  
        total += num  
    return total
```

Unit Tests

- Now to create the **first test** for our unit.
- We can perform a very basic test to see if our function will give us the intended result for a list with a single value.

```
def
test_list_add_with_one_number (self):
    # Arrange
    num_list = [5]
    # Act
    result = sum_list(num_list)
    # Assert
    self.assertEqual(result, 5)
```


Unit Tests

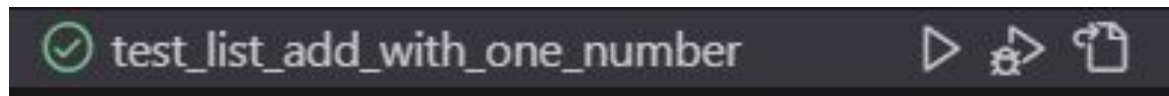
- Here is the full test class with our first test.

```
import unittest
from examples import sum_list

class TestCalculator(unittest.TestCase):
    def
test_list_add_with_one_number(self):
    # Arrange
    num_list = [5]
    # Act
    result = sum_list(num_list)
    # Assert
    self.assertEqual(result, 5)
```

Unit Tests

- We can now run the test and have a look at the result. For the first test we can see that our test has run without any failure.



- Let's make some more tests to see if our behaviour is in fact what we intend it to be.

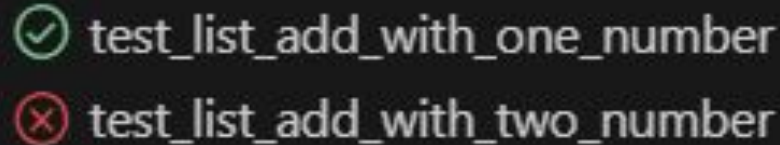
Unit Tests

- From our first test we saw that our behaviour return the single value when a list with one value is provided but what happens with two values in our list?

```
def
test_list_add_with_two_number (self):
    # Arrange
    num_list = [5, 10]
    # Act
    result = sum_list(num_list)
    # Assert
    self.assertEqual(result, 15)
```

Unit Tests

- Our **second test** has failed indicating there is an error in our code.



```
✓ test_list_add_with_one_number  
✗ test_list_add_with_two_number
```

- Let's take another look at our code to see what might have happened.

Unit Tests

- At closer inspection we can see that we have a small logical error that is preventing our test from passing.
- Remember when we correct this error all our previous tests should still pass.

```
def sum_list(num_list):  
    total = 0  
    for num in num_list:  
        total += num  
    return total
```

Logical error

Unit Tests

- We can fix our logical error and run our tests again to see if they all pass.

```
def sum_list(num_list):  
    total = 0  
    for num in num_list:  
        total += num  
    return total
```

✓ test_list_add_with_one_number

✓ test_list_add_with_two_number

Questions and Answers



Conclusion and Recap



Conclusion and Recap

- Unit Testing
 - **Process of testing** the behaviours of our program to make sure it behaves as intended.
- Arrange, Act, Assert
 - Pattern **used to structure** our unit tests.
- FIRST Principles
 - A **set of rules** we follow to create quick simple and accurate unit tests.

Questions and Answers



Thank you for attending



Department
for Education

