



# Welcome to this **Co**Grammar session: 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** midway and at the end of the session, should you wish to ask any follow-up questions. Moderators are going to be answering questions as the session progresses as well.
- 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](http://www.hyperiondev.com/support)
- Report a **safeguarding** incident:  
[www.hyperiondev.com/safeguardreporting](http://www.hyperiondev.com/safeguardreporting)
- We would love your **feedback** on lectures: [Feedback on Lectures](#)

# Skills Bootcamp

## 8-Week Progression Overview

### Fulfil 4 Criteria to Graduation

#### ✓ Criterion 1: Initial Requirements

- **Timeframe:** First 2 Weeks
- **Guided Learning Hours (GLH):**  
Minimum of 15 hours
- **Task Completion:** First four tasks

**Due Date: 24 March 2024**

#### ✓ Criterion 2: Mid-Course Progress

- **Guided Learning Hours (GLH): 60**
- **Task Completion:** 13 tasks

**Due Date: 28 April 2024**

# Skills Bootcamp Progression Overview

## ✓ Criterion 3: Course Progress

- **Completion:** All mandatory tasks, including Build Your Brand and resubmissions by study period end
- **Interview Invitation:** Within 4 weeks post-course
- **Guided Learning Hours:** Minimum of 112 hours by support end date (10.5 hours average, each week)

## ✓ Criterion 4: Demonstrating Employability

- **Final Job or Apprenticeship Outcome:** Document within 12 weeks post-graduation
- **Relevance:** Progression to employment or related opportunity



# CoGrammar

## Unit Testing

**SKILLS  
FOR LIFE**

**SKILLS BOOTCAMPS**



Department  
for Education

**April 2024**

# Learning Objectives

- Define unit testing
- Explain the use of unit testing in software engineering.
- Define arrange, act, assert pattern.
- Use arrange, act, assert pattern for structuring tests.
- Describe the FIRST principles
- Explain how following the FIRST 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.

# Unit Testing

- **Advantages:**
  - Catch errors early
  - Improve code quality
  - Refactor with confidence
  - Document code behavior
  - 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 behavior 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).



# 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
  - Fast
  - Independent
  - Repeatable
  - Self Validating
  - Thorough

# 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 in 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.

# Unit Tests





# Unit Tests

- Different packages for unit test. 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.

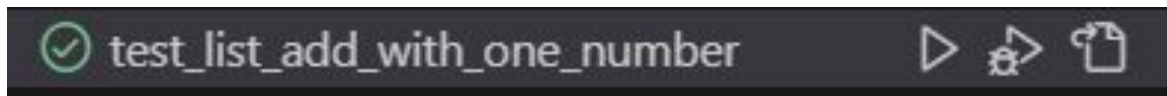
```
import unittest
from examples import sum_list

class TestExamples(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 we 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

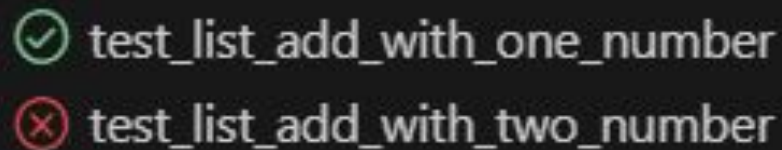
- From our first test we saw that our behaviour return the single value when a list with one value is provided but what happens with two values in our list?

```
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



**Let's take a short  
break**





# Summary





# Summary

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

# Questions and Answers



# Thank you for attending



Department  
for Education

CoGrammar

