



CoGrammar

Lecture 15: Unit Testing

**SKILLS
FOR LIFE**

SKILLS BOOTCAMPS



Department
for Education

Lecture Housekeeping

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly.
(FBV: Mutual Respect.)
- 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 Open Classes.
You can submit these questions here: [Open Class Questions](#)

Lecture 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](#)

Lecture Objectives

- Understand the importance of testing
- See how tests are implemented in Python
- Learn the techniques used for testing



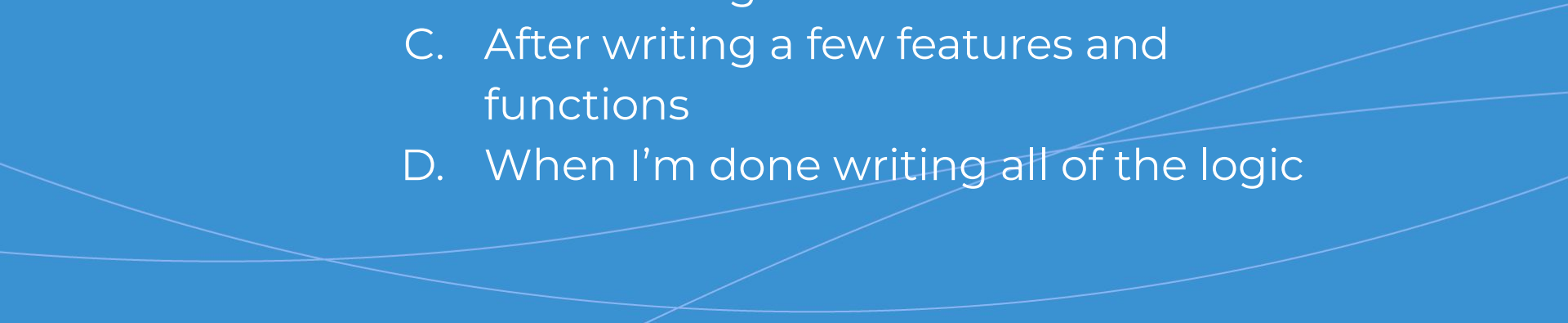
**How do you currently test you
code? (Answers in the questions
tab)**





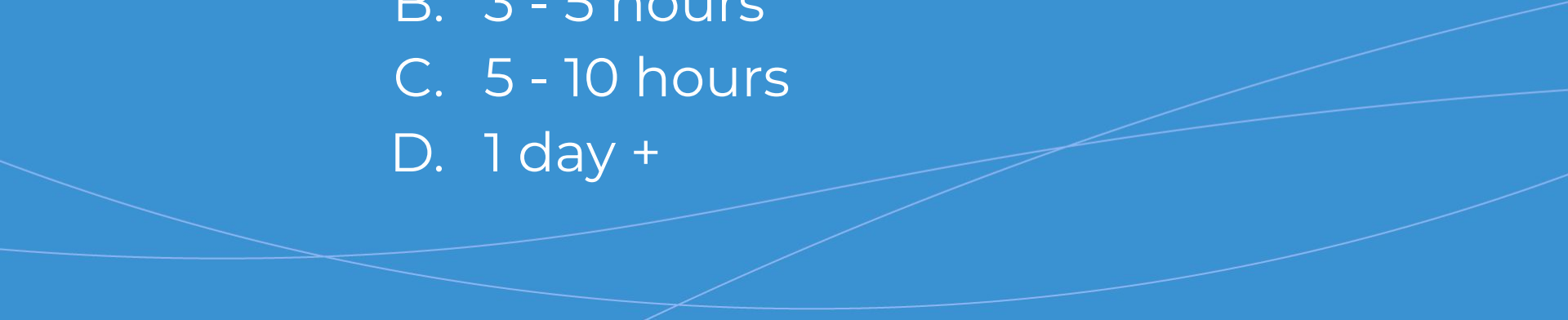
As you are writing your code, how often do you test it?



- A. After writing each line
 - B. After writing each function
 - C. After writing a few features and functions
 - D. When I'm done writing all of the logic
- 



What is the longest time you've spent debugging an application?

- A. 2 hours or less
 - B. 3 - 5 hours
 - C. 5 - 10 hours
 - D. 1 day +
- 

Testing

There are many ways that we can test the functionality of our code.

- Building dummy UIs
- Using print statements
- Writing each function in a separate file and putting it back in the main file when we are confident that it works.

Testing

These techniques may feel like quick quick ways to test the functionality of your code, have you ever taking into consideration:

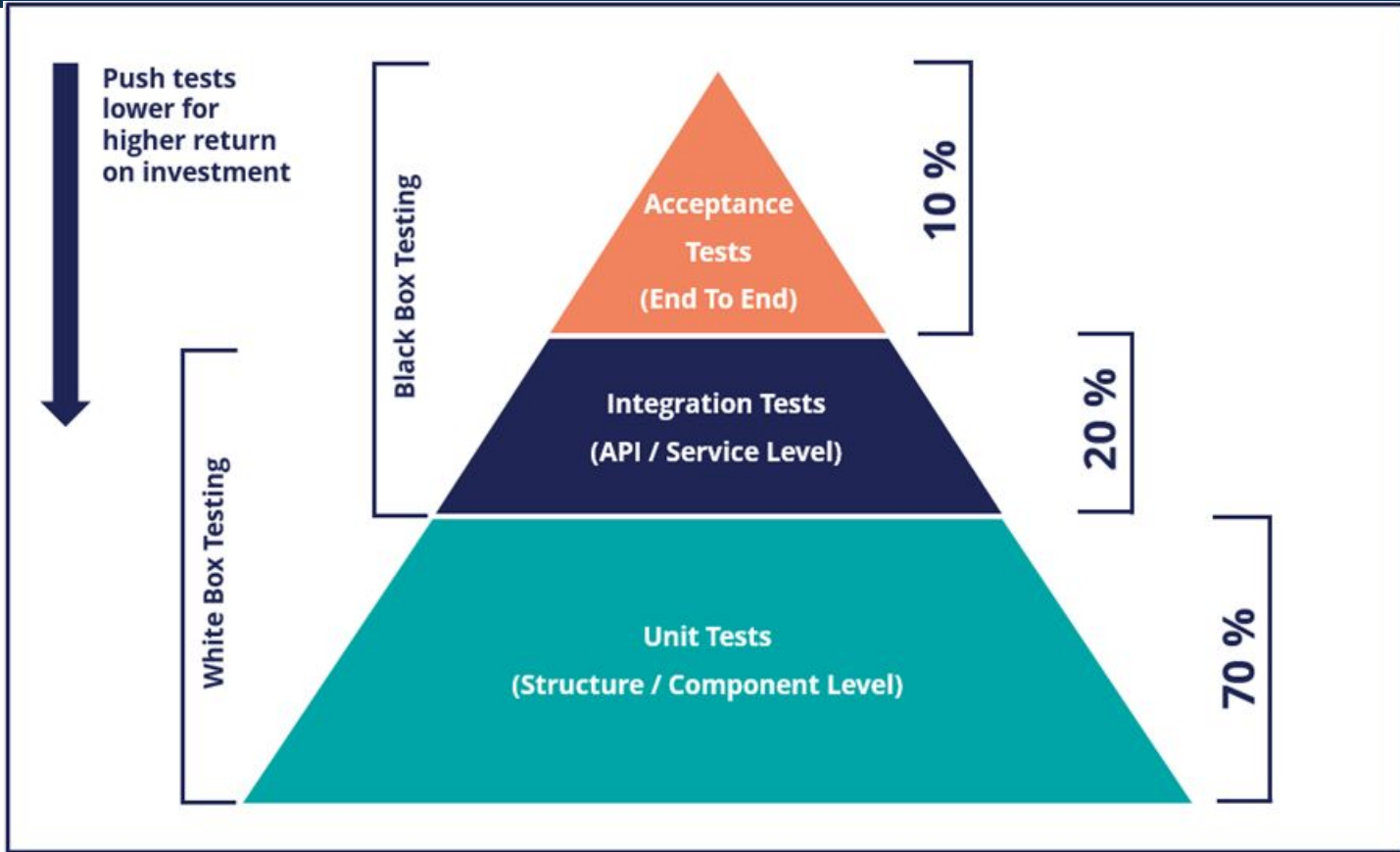
- How often do I need to create these 'quick' tests?
- How much of my development time is being taken to do theses tests?
- Do I really look forward to making these quick tests when I want to update the code?

Automated Testing

What is it

As the name suggests, automated testing is the process of automating the test process, traditionally, there are 3 levels to automated tests.

1. Unit Testing
2. Integration Testing
3. Acceptance Testing



Unit Testing

As we can see in the previous diagram, unit testing forms the base of our pyramid and is the most significant type of testing that we perform in our process.

What is it.

- Tests that are performed on a component level allowing use to test individual functionality like functions, classes or modules.
- Isolate each unit and ensures that each small piece is contributing to the whole

Unit Tests

Unit tests might seem a bit intimidating but it's important to remember that they are just code that is used to run other code.

How do they work:

1. We have the main code that we want to test
2. We make a unit test method
3. We call the main code in the unit test method.
4. We compare the result to what we expect to get
5. The test passes or fails

Benefits

- **Confidence** in our code, we can be confident that each function will perform as it should, and when making changes, we have something to confirm that the changes do not break our application.
- **Improved Code Quality**, Like all engineering, testing each components ensures that the whole system is well built
- **Enforces good coding practices**, If you are unable to test a method, most of the time it means that the function does not conform to certain programming principles. Tests encourage decoupling.

Prerequisites

- Understanding of Classes
- Understanding of modularization

To implement unit tests in Python, we will need a module that can handle our tests for us. We will be looking at the built in unittest module.

Setting Up Our Tests

Before we get into the core concepts around unit testing, let's see how we can set up our tests.

1. Create the file that will contain the code that we want to test
2. Create another file with the same name as the file that we want to test, but make sure it starts with 'test'
3. In the test file, import ``unittest``
4. Import the code file that will contain the functionality that you want to test
5. Create a class with the same name as the class in the file that you want to test (assuming there is a class), make sure to add 'Test' as the first word in the class name
6. Outside of the class, create a ``if __name__ == '__main__':`` and add ``unittest.main()``
7. We will write our test within the class as individual methods.

Setting Up Our Tests

```
# test_todo_list.py
import unittest
from todo_list import TodoList

class TestTodoList(unittest.TestCase):
    def setUp(self):
        # Create a new TodoList instance before each test
        self.todo_list = TodoList()

    def test_add_task(self):
        # Test if add_task method correctly adds a task
        self.todo_list.add_task("Task 1")
        self.assertEqual(self.todo_list.tasks, [])

    def test_update_task(self):
        # Test if update_task method correctly updates an existing task
        self.todo_list.add_task("Task 1")
        self.todo_list.update_task("Task 1", "Updated Task 1")
        self.assertEqual(self.todo_list.tasks, ["Updated Task 1"])

    def test_remove_task(self):
        # Test if remove_task method correctly removes a task
        self.todo_list.add_task("Task 1")
        self.todo_list.remove_task("Task 1")
        self.assertEqual(self.todo_list.tasks, [])

if __name__ == '__main__':
    unittest.main()
```

Principles of Unit Testing

To write our tests, there are two main things that we need to keep in mind.

AAA

This is the format that we should write our test functions in, it highlights the different sections that we need in order to effectively test our operations.

FIRST

These are the attributes that make up a good unit test and are important to keep in mind when creating our unit tests.

AAA

Arrange

- This is where we will set up our dependencies, if there are any
- We will set the values that we will need to pass as arguments to our methods

Act

- This is where we call the method that we would like to test
- If the function returns a value, we will store the result in a variable typically named 'actual'

Assert

- This is where we compare the 'actual' result to the result that we expect to get
- If the result does not meet our comparison, the test will fail

AAA

```
test_calculator.py × calculator.py
test_calculator.py > TestCalculator > test_add
1  import unittest
2  from calculator import Calculator
3
4  class TestCalculator(unittest.TestCase):
5      def setUp(self) -> None:
6          self.__calculator = Calculator()
7
8      def test_add(self):
9          # Arrange
10         num1 = 5
11         num2 = 10
12
13         # Act
14         actual = self.__calculator.add(num1, num2)
15
16         # Assert
17         self.assertEqual(15, actual)
18
```

FIRST

First defines characteristics that would make our unit tests good.

Fast

- Our tests should run fast (milliseconds)

Isolated

- Each test should be self contained and not dependent on any other test
- We should be able to run our tests in any order

Repeatable

- Tests should be consistent and give a predictable output everytime we run them
- We should avoid having logic that could fail in our tests\

Self-Validating

- We should have a clear pass and fail for our tests without manual intervention required.

Timely

- Tests should be written at the right time, preferably before we write the code.

Let's Breathe

Let's take a small break before moving on to the next topic.

Approach

Since unit tests just call other methods and functions, we can implement them in a number of ways.

- Adding tests to legacy code to help us make it more manageable in the future.
- We can implement tests when we notice that we have issues in the code and want to debug them
- We can use unit tests whenever we don't want to create makeshift UIs to test certain functions.

The major drawback of the approaches above is that they only consider using tests when we run into issues and they are not focused on preventing issues in the first place.

Red - Green - Refactor

We can use the RED-GREEN-REFACTOR approach to build our code out in a Test Driven Development way. But there are a few things to note:

- The process might seem cumbersome, but just think back to how long it takes you to resolve some simple bugs, this approach will save you all of that time, you just need to invest in the building process a bit more
- The process requires practice to get perfect and requires you to follow every single step in the process even if it feels unnecessary.

Red - Green - Refactor

Process

Before we start

1. We start of with two classes (each in their own file), one will eventually have the code that we want to test, the other will have the actual tests.
2. In the logic class,, you can have the methods that you are going to implement (based on your class diagrams), but you should add the **pass** keyword so that they do not throw errors as we will not implement the code just yet

Red - Green - Refactor

Red

- Pick a function that you want to test/create
- Create a method with the same name as that method but with `test` at the beginning
- Write a **single** test, that will test a **single** operations of the function
- Run the test and it should fail (if not, there is a problem with the test)

Green

- Write the minimum code required to make the test pass, write the first code that comes to your head and not the most optimal solution
- Run the test and it should pass

Refactor

- If the last step was good, optimize the code until you are satisfied
- If the test passes, you can start the process again

Red - Green - Refactor

Important Notes

- Test specific functionality
 - **One concern per test function,**
 - Eg, you want to create a function that subtracts values, but it should not return negative values
 - We test the functionality to subtract on its own
 - We create another test to check if we can get a negative value
- Write clear and descriptive tests
 - We should know what is being tested from the function name
 - The code should be concise and easy to read and maintain
- Test positive and negative scenarios
 - Users will find ways to break your application, make sure you are accounting for incorrect input and edge cases and not just desirable operations.
- Aim for high code coverage
 - Not all functions can be tested, but it's important to test as many functions as possible
 - It's important to also make sure that you have more than 1 test per function

Red - Green - Refactor

Benefits

- **Insight into requirements,** as you write the tests, you are gaining a better understanding of what the application needs
- **Saves hours, days and weeks of debugging,** as we write each function, we are ensuring that it does what we need it to do, allowing us to catch errors early and fix them
- **Gives us confidence when making changes,** If we need to change a feature, we can trust that our tests will inform us of whether the test breaks our functionality

CoGrammar

Q & A SECTION

**Please use this time to ask
any questions relating to the
topic, should you have any.**

Red - Green - Refactor

PRACTICAL



CoGrammar

Thank you for joining!