Welcome to "From Manual to Automated - An Introduction to Coding for QA Testers"! This section is designed for Manual Quality Assurance (QA) testers who are eager to transition from manual testing to automated testing using coding. We will explore the essential concepts and tools that will help you enhance the efficiency and effectiveness of your testing process. By the end of this tutorial, you will have a foundation in coding for QA testing and be ready to embark on your automation journey.

Transition to About Me in Slide

Transition to Introduction

Transition to Agenda

# Agenda:

1. Overcoming Initial Apprehensions about Automation
2. Why Automation is Essential
3. Introduction to Coding
4. Writing your first test script
5. Handling Test Data and Test Frameworks
6. Continuous Integration and Continuous Testing
7. Selecting the right tools/Setting up Environment
8. Q&A

Transition to Audience Background

# Background:

1. Familiarity with Manual Testing Concepts
2. Basic or non-basic knowledge of programming languages/concepts

Transition to first section of the lesson

# Overcoming Initial Apprehensions about Automation

I have compiled the challenges I have faced when initiating test automation to teams, or fellow QA testers themselves. Here are the examples:

**1. Lack of Automation Skills:**

- Challenge:(In my experience, there are colleagues that I noticed are very scared of moving towards automation, because) Many manual testers may not have the necessary skills or experience in automation tools and programming languages.

- **Solution:** Training programs, workshops, and online resources can help bridge this gap. Encouraging team members to learn automation skills or hiring automation specialists can also be beneficial.

2. **Resistance to Change:**

- Challenge: Team members may be resistant to adopting automation due to fear of job loss, unfamiliarity with automation tools, or a preference for manual testing.

- Solution: Open communication about the benefits of automation, addressing concerns, and demonstrating how automation complements manual testing rather than replacing it can help overcome resistance.

3. **Tool Selection:**

- Challenge: Choosing the right automation tools can be challenging due to the variety of options available, each with its own strengths and weaknesses.

- Solution: Conduct thorough research, evaluate tools based on team needs, scalability, and community support. Piloting different tools and getting feedback from the team can aid in making an informed decision.

4. **Limited Budget:**

- Challenge: Budget constraints may limit the acquisition of automation tools or training resources.

- Solution: Prioritize automation efforts based on critical test scenarios, explore open-source tools, and leverage free training resources. Building a business case that highlights the long-term cost savings can also help secure budgetary support.

5. **Test Environment Setup:**

- Challenge: Configuring and maintaining test environments for automation can be complex and time-consuming.

- Solution: Invest time in creating reproducible and stable test environments. Tools like containerization can help in managing dependencies and ensuring consistency across different environments.

6. **Continuous Integration/Continuous Deployment (CI/CD) Integration:**

- Challenge: Integrating automated tests into the CI/CD pipeline may be a hurdle, especially if the team is new to these practices.

- Solution: Collaborate with the development and DevOps teams to seamlessly integrate automated tests into the CI/CD pipeline. Provide training on CI/CD concepts if necessary.

7. **Test Data Management:**

- Challenge: Creating and managing test data for automated tests can be a complex task.

- Solution: Develop strategies for effective test data management. This may include using tools for data generation, maintaining separate test databases, or leveraging APIs for data setup.

8. **Maintenance Overhead:**

- Challenge: Automated tests require maintenance as the application evolves, which can be time-consuming.

- Solution: Design automation scripts with maintainability in mind, conduct regular code reviews, and update scripts as needed. Use version control systems to manage changes and collaborate effectively.

9. **Measuring Automation ROI:**

- Challenge: Demonstrating the return on investment (ROI) of automation efforts can be challenging.

- Solution: Define clear metrics for success, such as reduced testing time, increased test coverage, and fewer post-release defects. Regularly track and report these metrics to showcase the impact of automation.

10. **Incomplete Test Coverage:**

- Challenge: Automating all test scenarios may not be feasible, leading to incomplete test coverage.

- Solution: Prioritize test scenarios based on business criticality and impact. A risk-based testing approach can help identify and automate the most important test cases first.

Transition to Why Automation is Essential

# Why Automation is Essential

*Accelerated Testing Cycles*

* Automation significantly speeds up the testing process, enabling quicker releases.
* Rapid feedback loops empower teams to iterate and improve swiftly.

*Improved Test Accuracy and Consistency*

* Automated tests perform tasks precisely, reducing the chance of human errors.
* Consistency in test execution ensures reliable results across environments.

*Enhanced Test Coverage*

* Automation allows for extensive test coverage, ensuring critical scenarios are thoroughly tested.
* Comprehensive testing contributes to better software quality.

*Resource Optimization*

* Automated tests can run 24/7, maximizing resource utilization.
* Reduces dependency on manual testers for repetitive tasks.

**When to Consider Automation?**

*Repetitive Test Scenarios*\*

* Consider automation when there are repetitive test cases that need frequent execution.

*Large and Complex Applications*\*

* Automation shines in testing large and intricate software systems.

*Frequent Code Changes*\*

* In dynamic development environments, where code changes are frequent, automation aids in quick validation. Ensures that new features don't break existing functionalities.

*Performance and Load Testing*\*

* Automation is indispensable for simulating large user loads and stress testing. Identifying performance bottlenecks and ensuring system stability.

Transition to Coding

# Introduction to Coding:

## 1. Variables:

Variables are like containers that store information. They hold data that can be used and changed throughout the program.

Analogy: Compare variables to labeled boxes in which you can store different things. Each box (variable) has a name (identifier) that you can use to access its contents.

## 2. Data Types:

Describe that data types define the kind of data a variable can hold, such as numbers, text, or true/false values.

Python is strongly typed; You don't have to specify what's going into the container when you create it. However, once you put something inside, the container doesn't like sudden changes. If you initially put a number in there, you can't later decide to put a word without the computer complaining. It's a bit more strict about what can go in the container.

Example – if search\_query is a string, you cannot perform arithmetic operators obviously because this is a string, and result\_count is an integer. You cannot perform string operators as well if a variable is a integer data type.

## 3. Operators:

Explain that operators are symbols or keywords that perform operations on variables and values.

- Examples:

- Arithmetic Operators: Addition (+), Subtraction (-), Multiplication (\*), Division (/).

- Comparison Operators: Equal to (==), Not equal to (!=), Greater than (>), Less than (<).

- Logical Operators: AND (&&), OR (||), NOT (!).

## 4. Control Flow Structures

Control flow structures are like decision-making and looping mechanisms that control the order in which instructions are executed in a program.

If Statement (Decision-making):

- way to make decisions in the program based on conditions.

- \*\*Example:\*\*

```plaintext

if (condition) {

// Code to be executed if the condition is true

}

```

### Else Clause (Alternative Decision):

- "else" clause is an alternative path when the initial condition is not met.

- \*\*Example:\*\*

```plaintext

if (condition) {

// Code to be executed if the condition is true

} else {

// Code to be executed if the condition is false

}

```

### Else If (Multiple Conditions):

- Introduce "else if" to check multiple conditions in sequence.

- \*\*Example:\*\*

```plaintext

if (condition1) {

// Code to be executed if condition1 is true

} else if (condition2) {

// Code to be executed if condition2 is true

} else {

// Code to be executed if none of the conditions are true

}

```

### Loops:

loops are a way to repeat a block of code.

- \*\*For Loop Example:\*\*

```plaintext

for (initialization; condition; update) {

// Code to be executed in each iteration

}

```

- \*\*While Loop Example:\*\*

```plaintext

while (condition) {

// Code to be executed as long as the condition is true

}

```

## Functions and Modularizations

### Functions:

- A function is a named, reusable block of code that performs a specific task. It allows you to break down a program into smaller, manageable pieces.

### Parameters and Return Values:

- Functions can take parameters (inputs) and return values (outputs). Parameters allow the function to receive external data, and return values allow it to provide a result.

### Modularization:

- Modularization is the practice of breaking a program into smaller, independent modules or functions. It helps improve code organization, readability, and maintainability.

### Benefits of Modularization:

- Reusability: Emphasize that once a function is defined, it can be reused in different parts of the program.

- Readability: Breaking code into smaller modules makes it easier to understand and maintain.

- Maintenance: Changes or updates can be made to individual modules without affecting the entire program.

# Class and Method

In object-oriented programming (OOP), the notation `class.method` typically refers to invoking a method (function) that belongs to a specific class. Here's a breakdown of what this notation means:

## Class:

In OOP, a class is a blueprint for creating objects. It defines a set of attributes and methods that objects created from the class will have. Think of a class as a template or a prototype.

## Method:

A method is a function that is associated with a class. It defines the behavior of objects created from the class. Methods can be called on instances of the class, and they can perform actions or provide functionality related to the class.

In QA:

Class:

- In Quality Assurance (QA), a class is like a predefined set of rules or instructions for creating a specific type of test scenario or test case. It outlines what should be tested and how to test it.

Attributes:

- Attributes in QA terms could be the various aspects or characteristics of the system or application that you are testing. These might include things like input data, expected outcomes, or specific conditions under which the testing occurs.

Methods:

- Methods, in QA, could be thought of as the specific actions or steps that need to be taken during the testing process. For example, a method could be a series of steps to navigate through a website, input data into a form, and verify the results.

Objects:

- In QA, an object could be considered as an instance of a test scenario or test case. It's a specific occurrence where the predefined rules (class) are applied to a particular situation to determine if the system behaves as expected.

Blueprint or Recipe:

- Think of the class as a blueprint or recipe for a specific type of test. It outlines the structure, steps, and expected outcomes, much like a recipe guides you through a cooking process.

So, in QA terms, a class is a defined structure that specifies what to test, how to test it, and what the expected results should be, forming the basis for creating and executing tests.

Certainly, let's explain `class.method` in the context of QA:

\*\*Class:\*\*

- In Quality Assurance (QA), a class can be thought of as a reusable set of instructions or guidelines for performing a specific type of test. It defines what needs to be tested, how to test it, and what the expected outcomes are.

\*\*Method:\*\*

- A method in QA is like a specific action or step within a test. It represents a part of the overall testing process. For example, a method could be a series of steps to interact with a user interface, input data, or verify the correctness of a particular functionality.

\*\*Class.Method:\*\*

- When we say `class.method`, it means applying a specific set of testing instructions (class) to perform a particular action or verification (method). It's like following a predefined set of rules to conduct a specific part of the testing process.

\*\*Example:\*\*

Suppose we have a class called `LoginFormTest`:

```python

class LoginFormTest:

def test\_successful\_login(self):

# Steps to input valid credentials and verify successful login

pass

def test\_invalid\_credentials(self):

# Steps to input invalid credentials and verify error message

pass

```

Here, `LoginFormTest` is the class, and `test\_successful\_login` and `test\_invalid\_credentials` are methods. Calling `LoginFormTest.test\_successful\_login()` means executing the steps defined in that method to test the successful login scenario.

In QA terms, `class.method` is about organizing and executing specific tests by following a predefined set of instructions, ensuring consistency and repeatability in the testing process.

# Libraries

A library is like your go-to toolkit, offering ready-made solutions and saving you time in your manual testing work. It's a collection of efficient tools that you can borrow and use without delving into the technical complexities behind them.

You don’t have to reinvent the wheel, because these are ready-made solutions.

Certainly! Here's a rewritten explanation:

### In Our Example, Using Selenium as Our Tool:

Let's picture Selenium as our testing tool, and Selenium comes with its own library. This library is like a set of pre-built instructions that we can use without having to dive into the details of coding every single action when interacting with a web browser.

1. Selenium as the Handy Tool:

- Selenium, in our case, is like a versatile tool in our testing toolkit. Instead of crafting detailed code for every action, Selenium provides a library of ready-to-use commands that simplify tasks like opening a web browser and performing various functions within it.

2. Pre-Built Functions for Web Browsing:

- Within Selenium's library, there are pre-built functions for common tasks like navigating to a website, filling out forms, or clicking buttons. We can leverage these functions without needing to create them ourselves.

3. No Need to Code Everything from Scratch:\*\*

- With Selenium's library, we don't have to start coding from scratch every time we want to interact with a web browser. It's like having a set of well-defined shortcuts that handle the complexities of browser interactions without us having to figure out the inner workings.

\*\*4. Simplifying Browser Interaction:\*\*

- Selenium's library simplifies our interaction with web browsers. We can call upon these pre-built functions, allowing us to focus more on designing effective test cases and less on the intricacies of coding browser-related actions.

\*\*5. Ready-Made Commands for Efficiency:\*\*

- We can efficiently perform actions like clicking links, verifying content, or navigating between pages by using the ready-made commands available in Selenium's library. It's akin to having a set of reliable, tried-and-tested instructions at our disposal.

\*\*6. Specific Tools for Specific Tasks:\*\*

- Selenium's library provides specific tools (functions) for specific tasks related to web browsing. We don't have to worry about the underlying code; we can simply use the appropriate tool for the job.

\*\*7. Enhancing Test Case Design:\*\*

- By utilizing Selenium's library, we enhance our ability to design effective test cases. We can focus on the expected behavior of the application under test rather than getting bogged down in the details of browser interaction.

In summary, Selenium's library acts as a valuable resource in our testing efforts, providing us with a set of predefined commands and functions. This simplifies our work, allowing us to efficiently interact with web browsers without the need to delve deeply into the coding intricacies.

# QA Perspective on Test Script and Coding

In a QA (Quality Assurance) perspective, the terms "Test Cases," "Test Scripts," and "Algorithms" have specific meanings and roles. Let's explore each one:

1. Test Cases:

- A test case is a detailed set of conditions or variables under which a tester will determine whether an application, system, or feature is working as intended. It includes input data, execution steps, and expected outcomes.

2. Test Scripts:

- A test script is a sequence of instructions written in a programming language (e.g., JavaScript, Python) to automate the execution of test cases. Test scripts can be manual (executed by a tester) or automated (executed by testing tools).

- QA Perspective:

- They are valuable for regression testing, where repetitive tests need to be executed to ensure that new changes do not break existing functionality.

3. Algorithms:

- An algorithm is a step-by-step procedure or formula for solving a problem or accomplishing a specific task. In software development and testing, algorithms are used to define the logic and process for various operations.

- QA Perspective:

- In testing, algorithms may be involved in the development of test data generation processes or in defining complex test scenarios.

- For performance testing, algorithms might be used to simulate user behavior or generate realistic load patterns.

- Functions/Modularization

Practical Example:

Consider a scenario where a QA tester is tasked with testing an e-commerce website's checkout process:

-Test Case:

- Define a test case to validate the successful completion of a purchase, including steps like adding items to the cart, entering shipping information, and confirming the order.

- Test Script:

- If automated testing is employed, a test script written in a programming language (e.g., Python) could automate the steps outlined in the test case.

-Algorithm:

- An algorithm might be used to generate various scenarios, such as testing different payment methods, checking how the system handles discounts, or simulating high traffic during the checkout process.

In summary, test cases define what needs to be tested, test scripts automate the execution of those test cases, and algorithms can be employed to enhance testing processes and scenarios in various ways.

# Test Frameworks:

Both Pytest and Unittest are capable testing frameworks, and the choice between them often depends on personal preference, project requirements, and the testing ecosystem in use. Pytest is known for its simplicity, readability, and powerful features, while Unittest is part of the Python standard library, making it readily available without additional installations.

## Pytest vs. Unittest:

### Syntax and Readability:

- Pytest:

- Pytest has a simpler and more readable syntax. Test functions are simple Python functions, and assertions are expressed using natural language constructs.

- Unittest:

- Unittest follows a more verbose syntax. Test cases are defined by creating classes that inherit from `unittest.TestCase`, and test functions must start with the word "test."

### Fixture Mechanism:

- Pytest:

- Pytest provides a powerful fixture mechanism, allowing the setup and teardown of resources before and after tests. Fixtures can be shared across multiple test functions or modules.

- Unittest:

- Unittest also supports fixtures but requires more boilerplate code for setup and teardown. Fixtures are typically implemented using `setUp()` and `tearDown()` methods in test classes.

### Test Discovery:

-Pytest:

- Pytest automatically discovers and runs all files matching the `test\_\*.py` or `\*\_test.py` naming pattern. No explicit test suite is required.

- Unittest:

- Unittest requires the creation of a test suite and explicit test discovery. This can be achieved using the `TestLoader` or `discover` module.

### Parameterized Testing:

- Pytest:

- Pytest supports parameterized testing, allowing the same test function to be called with different sets of parameters.

- Unittest:

- Parameterized testing in Unittest usually involves creating separate test methods for each set of parameters, resulting in more code duplication.

### Assertions:

- Pytest:

- Pytest's assert statements are more informative, providing detailed information on test failures.

- Unittest:

- Unittest uses the standard Python `assert` statement, which provides less detailed failure messages compared to Pytest.

### Parallel Test Execution:

- Pytest:

- Pytest supports parallel test execution out of the box, improving test execution speed.

- Unittest:

- Unittest does not have built-in support for parallel test execution, but third-party solutions can be integrated.

### Community and Ecosystem:

- Pytest:

- Pytest has a large and active community with a rich ecosystem of plugins and extensions.

- Unittest:

- Unittest is part of the Python standard library and has a well-established but more limited ecosystem.

### Test Reports:

- Pytest:

- Pytest generates detailed and informative test reports by default.

- Unittest:

- Unittest provides basic test reports, and additional tools may be needed for more detailed reporting.

# Introduction to Continuous Integration (CI) and Continuous Testing (CT) for QA:

## Continuous Integration (CI):

Continuous Integration is a software development practice where changes made to the source code of an application are automatically integrated and tested frequently. The primary goals of CI are to identify and address integration issues early in the development process, enabling teams to deliver higher-quality software more rapidly.

## Key Components of CI:

### Automated Builds:

CI systems automate the process of building the application from source code. This ensures that the application can be consistently and reproducibly built.

### Automated Testing:

Automated tests, including unit tests and integration tests, are executed as part of the CI process. This helps in identifying defects and regressions early in the development cycle.

### Version Control Integration:

CI systems are tightly integrated with version control systems (e.g., Git, SVN). They monitor repositories for changes and trigger builds and tests automatically when changes are detected.

### Continuous Feedback:

CI provides rapid feedback to developers about the health of the codebase. If a build or test fails, developers are immediately notified, enabling quick resolution of issues.

## Continuous Testing (CT):

Continuous Testing is an extension of CI that emphasizes the automated testing aspect throughout the entire software development lifecycle. It aims to ensure that every change made to the codebase is automatically validated through a comprehensive suite of tests.

## Key Components of CT:

### Automated Regression Testing:

CT includes automated regression tests that validate the existing functionality of the application after each code change. This helps in preventing the introduction of new defects.

### Parallel Test Execution:

To expedite the testing process, CT often involves running tests in parallel, enabling faster feedback and reducing testing cycle time.

### Performance Testing:

Continuous Testing may include performance testing to ensure that the application meets performance and scalability requirements as changes are introduced.

### Security Testing:

Security testing is integrated into the CT process to identify and address security vulnerabilities early in the development lifecycle.

### Shift-Left Testing:

CT promotes the concept of "shift-left," encouraging the early involvement of testing activities in the development process to catch issues as early as possible.

## Benefits for QA:

### Early Defect Detection:

CI/CT practices help in detecting defects early in the development process, reducing the cost and effort of fixing issues.

### Consistent Build and Deployment:

QA teams work with consistent builds, ensuring that the testing environment is stable and reliable.

### Faster Feedback:

Rapid feedback from automated tests allows QA teams to quickly assess the impact of changes and make informed decisions.

### Improved Collaboration:

CI/CT fosters collaboration between development and QA teams, encouraging shared responsibility for quality.

### Efficient Release Process:

With automated testing and continuous integration, the release process becomes more efficient, enabling faster and more reliable software delivery.

Why Return Values?

In a testing context, understanding why functions return values is crucial for verifying the behavior of the system under test. Let's break it down in the context of a test case:

1. \*\*Verification of Expected Results:\*\*

- Test cases often involve executing specific actions or operations and then checking if the system behaves as expected. Functions returning values allow you to capture the system's response and compare it against the expected outcome. For example, if you have a function that submits a form, it might return a message like "Form submitted successfully." In your test case, you can check if the actual result matches the expected result.

2. \*\*Checking Preconditions and Postconditions:\*\*

- Functions often set up certain conditions or states (preconditions) and then perform an action or operation. The returned value can indicate whether the precondition was met or if the action was successful (postcondition). For instance, a function that logs in might return a boolean value indicating whether the login was successful. In your test case, you can use this information to check if the system is in the expected state after the login attempt.

3. \*\*Data Validation:\*\*

- Functions that interact with data often return values, and these values can be crucial for data validation in your test cases. For example, a function that retrieves information from a database might return the actual data. Your test case can then validate if the retrieved data matches the expected data.

4. \*\*Error Handling:\*\*

- Test cases should also cover scenarios where things go wrong. Functions returning values, especially error codes or messages, allow you to verify how the system handles errors. For example, if a function encounters an invalid input, it might return an error message. Your test case can check if the system responds appropriately to such errors.

5. \*\*Chaining Test Steps:\*\*

- Test cases often consist of multiple steps or actions. Functions returning values enable you to chain these steps together. The result from one function can serve as input for the next, allowing you to create a sequence of actions that mimic real-world user interactions.

In summary, in a test case, functions returning values provide the necessary feedback and information for validating the system's behavior, making decisions based on test results, and ensuring that the system under test meets the specified requirements.