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

In my previous experiences, I faced many challenges when implementing or initiating test automation for teams. The outcome was either success or failure. I have compiled the top 5 challenges I encountered while introducing test automation to teams or fellow QA testers. Here are some 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.

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.

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.

# 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:

Python

- high level, interpreted programming language, known for its readability and versatility.

It is widely used for machine learning, artificial intelligence, automation and web development.

It uses whitespace indention (unlike all other languages, they emphasize on the usage of ‘;’). Format is uncluttered, and often uses English keywords. Also, it doesn’t use curly brackets to delimit blocks. Semicolons are used after statements.

Python places importance on Code Readability and Simplicity.

## 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.c

- \*\*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.~~

### ~~(In a Function, there are)~~

### ~~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.~~

~~- eadability: 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:

- An object is an instance of a class, representing a specific occurrence or scenario you want to test. It's the application of the class rules to a particular situation to see if the system behaves correctly.

**Example: In the context of Google Search testing, an object could be a specific search scenario you want to test, like searching for "Software Testing" on Google. The object, in this case, is the actual process of performing a search for a particular term. You want to see if the search functionality behaves correctly for this specific scenario, making sure that the system displays relevant results.**

Example: An "LoginPage" object could be a specific test case where a user enters valid credentials and logs in.

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.

Putting it all together:

* You have a class (e.g., "LoginPage") that defines attributes (elements on the page) and methods (actions to perform on the page).
* An object is then created based on this class, representing a specific test scenario (e.g., logging into a web application).
* During testing, you apply the methods defined in the class to the object to interact with the system and verify if it behaves as expected.

**In summary, in QA, classes, attributes, methods, and objects are used to structure and organize testing scenarios. Classes provide a blueprint, attributes represent characteristics, methods define actions, and objects are instances of specific test scenarios where you apply these rules to validate the system's behavior.**

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

\*\*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

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. It's a collection of efficient tools that you can borrow and use without delving into the technical complexities behind them.

In our case, we are going to use selenium.

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

In a typical use case, you might find an input element (like a search box), use `send\_keys` to type a search query, and then use `click` on the search button to initiate the search. These methods are fundamental for interacting with web elements during automated testing with Selenium.

By - is a class in Selenium that provides mechanisms to locate elements on a web page. It is often used in conjunction with the **find\_element** method to specify how to locate a particular element.

1. \*\*`find\_element` method:\*\*

- This Selenium method is used to locate a single web element on a webpage based on a specified criterion such as ID, name, class name, XPath, or CSS selector.

- Example: `search\_box = driver.find\_element("name", "q")` - This finds an input element with the name attribute "q" and assigns it to the `search\_box` variable.

2. \*\*`send\_keys` method:\*\*

- Once a web element is located using `find\_element`, `send\_keys` is used to simulate typing or entering text into that element.

- Example: `search\_box.send\_keys("Software Testing")` - This types the text "Software Testing" into the input element referenced by the `search\_box` variable.

3. \*\*`click` method:\*\*

- The `click` method is used to simulate a mouse click on a web element. It is commonly used for buttons, links, or any clickable elements.

- Example: `search\_button.click()` - This simulates clicking on the element referenced by the `search\_button` variable.

## DOM – Document Object Model

It is also known as the Document Object Model, which represents the entire HTML or XML Document. It also serves as our interface to access all elements that we need to interact with.

Google Example: - show them how to access the elements.

When looking at the google search page, we see text, links, textboxes, buttons, logos. How do we interact with these?

We access the DOM. Right click on an element and select Inspect element. Every element has its own unique name. To access them, we usually look at the names of an element. Example, the search box name is name="q". The button name is btnk.

Show them the code:

IN CASE THERES QUESTIONS:

In the context of web development and testing, DOM stands for Document Object Model. The DOM is a programming interface for web documents that represents the structure of a document as a tree of objects. Each object corresponds to a part of the document, such as elements, attributes, and text.

Here's a breakdown of key concepts related to the DOM for a QA tester:

1. **Document:**
   * In the DOM, a document represents the entire HTML or XML document. It serves as the entry point to the content and structure of the web page.
2. **Object Model:**
   * The DOM is an object-oriented representation, meaning that elements in the document are represented as objects. These objects can be manipulated using programming languages like JavaScript.
3. **Tree Structure:**
   * The DOM organizes the document as a tree structure, where each node represents an element, attribute, or piece of text. The relationships between nodes define the hierarchy of the document.
4. **Nodes:**
   * Nodes are the individual objects in the DOM tree. There are different types of nodes, such as Element nodes (representing HTML elements), Attribute nodes (representing attributes of elements), and Text nodes (representing text content).
5. **Manipulation:**
   * One of the primary purposes of the DOM is to enable dynamic manipulation of the document. QA testers may need to interact with the DOM to test functionalities such as form submissions, element visibility, or content updates.
6. **Event Handling:**
   * The DOM is crucial for handling events on a web page. This includes actions like clicks, keypresses, or form submissions. QA testers may need to verify that event handlers are working as expected.
7. **Cross-Browser Compatibility:**
   * QA testers should be aware of potential differences in how browsers implement the DOM. Testing across multiple browsers helps ensure that web applications work consistently for all users.
8. **Browser Developer Tools:**
   * Testers often use browser developer tools to inspect and manipulate the DOM during testing. These tools provide insights into the current state of the DOM and assist in debugging.

Understanding the DOM is essential for QA testers, especially when dealing with dynamic web applications where elements change or update based on user interactions. Testing the interactions and behaviors driven by the DOM ensures a seamless user experience.

Class (GoogleSearchTest):

The class GoogleSearchTest is a blueprint that defines a set of instructions for testing Google search functionality.

Attributes (self.driver):

self.driver is an attribute of the class. It represents an instance of the Selenium WebDriver (webdriver.Chrome()), allowing interaction with the web browser.

Methods (\_\_init\_\_, navigate\_to\_google, perform\_search, close\_browser):

\_\_init\_\_: This method is a constructor that initializes the class. It creates an instance of the Chrome driver (self.driver) when an object of the class is created.

navigate\_to\_google: This method navigates the browser to Google.com.

perform\_search: This method performs a search on Google by locating the search input element, entering a search query, and clicking the 'Google Search' button.

Validate\_search\_results: it performs a validation if the search results are what we expect.

close\_browser: This method closes the browser, releasing resources.

Object (google\_test):

google\_test is an object instantiated from the GoogleSearchTest class. It represents a specific test scenario where a user navigates to Google,

performs a search, validates the result and closes the browser.

Putting It Together:

The class (GoogleSearchTest) provides a structure with methods and attributes for testing Google search functionality.

The object (google\_test) is an instance of the class, representing a specific test case or scenario.

Methods of the class are applied to the object (google\_test) to execute the steps of the test scenario.

In summary, this code demonstrates the concepts of class, attributes, methods, and objects in the context of testing Google search using Selenium.

The class defines the testing instructions, attributes represent elements to interact with, methods perform actions, and the object is an instance of a specific test scenario.

# QA Perspective on Test Script and Coding

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.

# Transition to Coding

Checkbox:

# '''

# Class (TestCheckbox):

# - TestCheckbox is a class that represents a set of tests related to checkbox functionality.

# Attributes (self.driver):

# - self.driver is an attribute of the class, representing the Selenium WebDriver (webdriver.Chrome()). It is used to interact with the web browser.

# Methods (setup\_method, teardown\_method, test\_toggle\_checkboxes):

# - setup\_method: This method sets up the test environment by creating a new instance of the Chrome driver and navigating to

# the specified URL (https://the-internet.herokuapp.com/checkboxes).

# - teardown\_method: This method is called after the test execution and closes the browser (self.driver.quit()), releasing resources.

# - test\_toggle\_checkboxes: This method contains the actual test scenario. It finds the checkboxes on the page, interacts with them, and asserts the expected conditions.

# Test Execution:

# The test is executed by creating an object of the TestCheckbox class and invoking its methods.

# The setup\_method method is called to set up the test environment.

# The test\_toggle\_checkboxes method is executed, performing the checkbox toggling test.

# The teardown\_method method is called to clean up after the test.

# Putting It Together:

# The class provides a structure for testing checkbox functionality.

# Attributes (self.driver) represent the WebDriver instance.

# Methods (setup\_method, teardown\_method, test\_toggle\_checkboxes) define the testing process.

# Test execution involves creating an object of the class and invoking its methods.

# In summary, this code demonstrates the structure of a test class for checkbox functionality using Selenium.

# It includes setup, teardown, and an actual test method to interact with checkboxes on a web page.

# '''

# 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 Git

# ~~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.~~