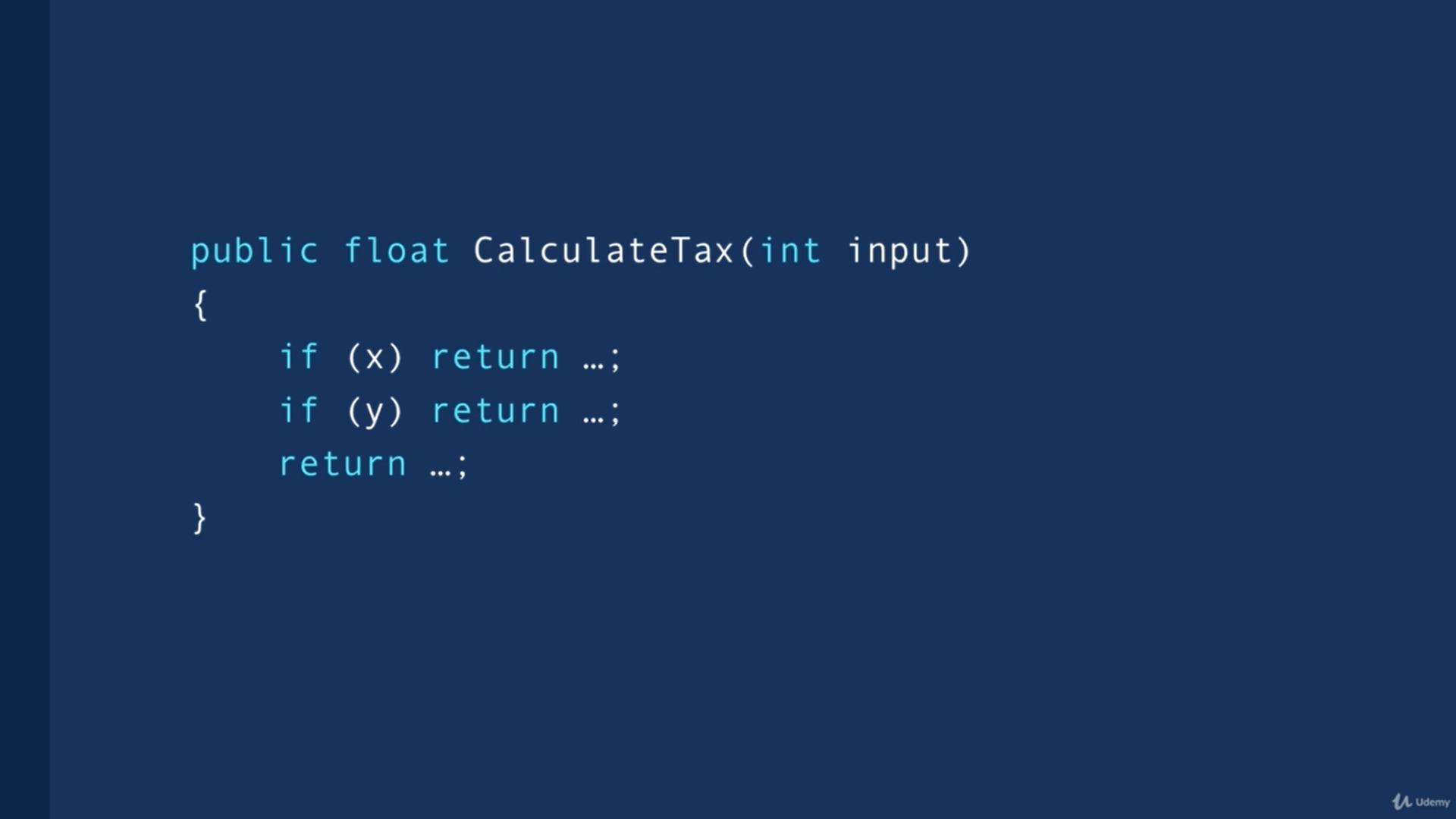
1. What is automated testing?

Automated testing is the practice of writing code to test our code, and then run those tests in an automated fashion. With automated testing, our source code consists of application code, which we also call production code and test code.

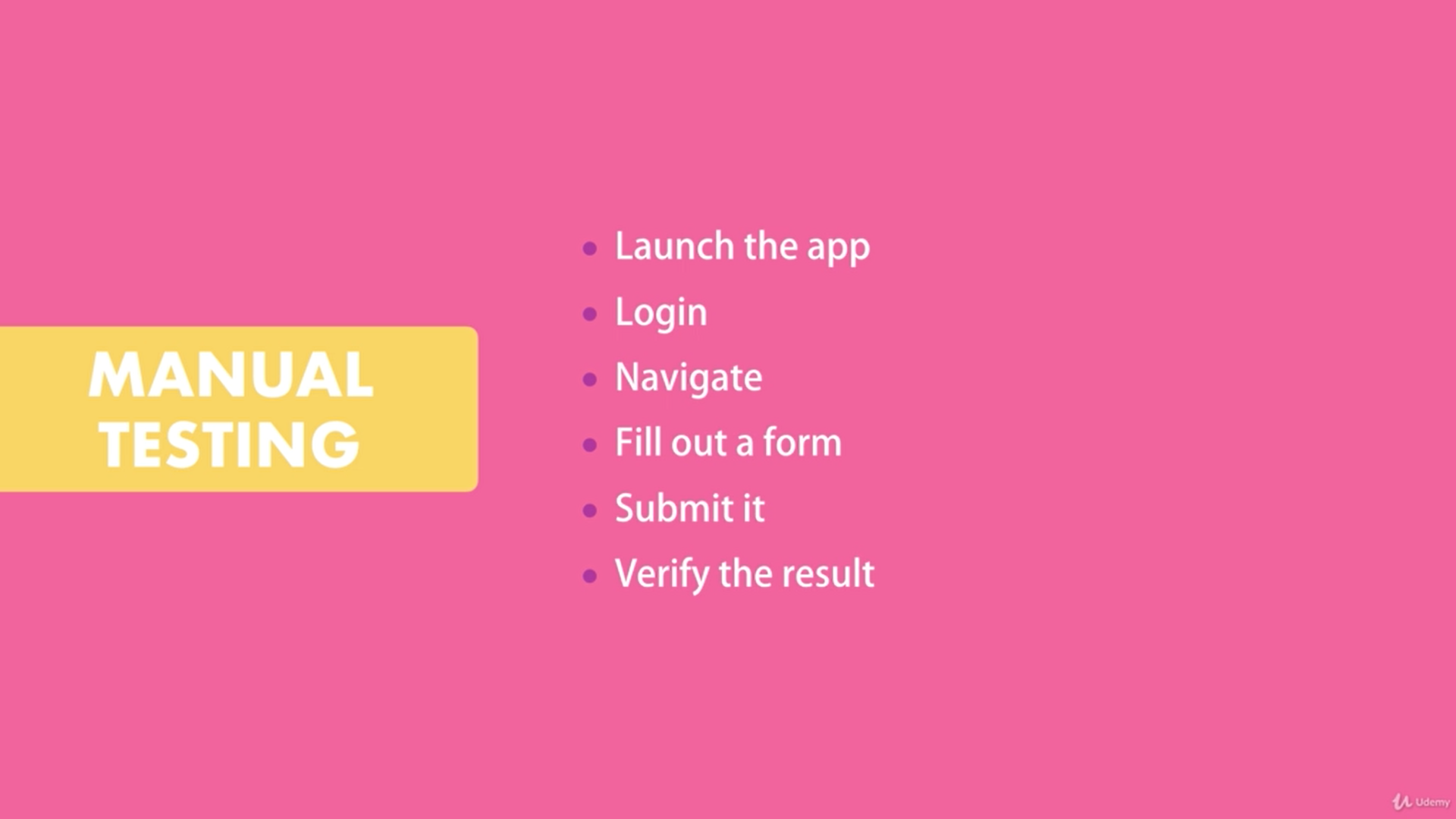


Here is an example:

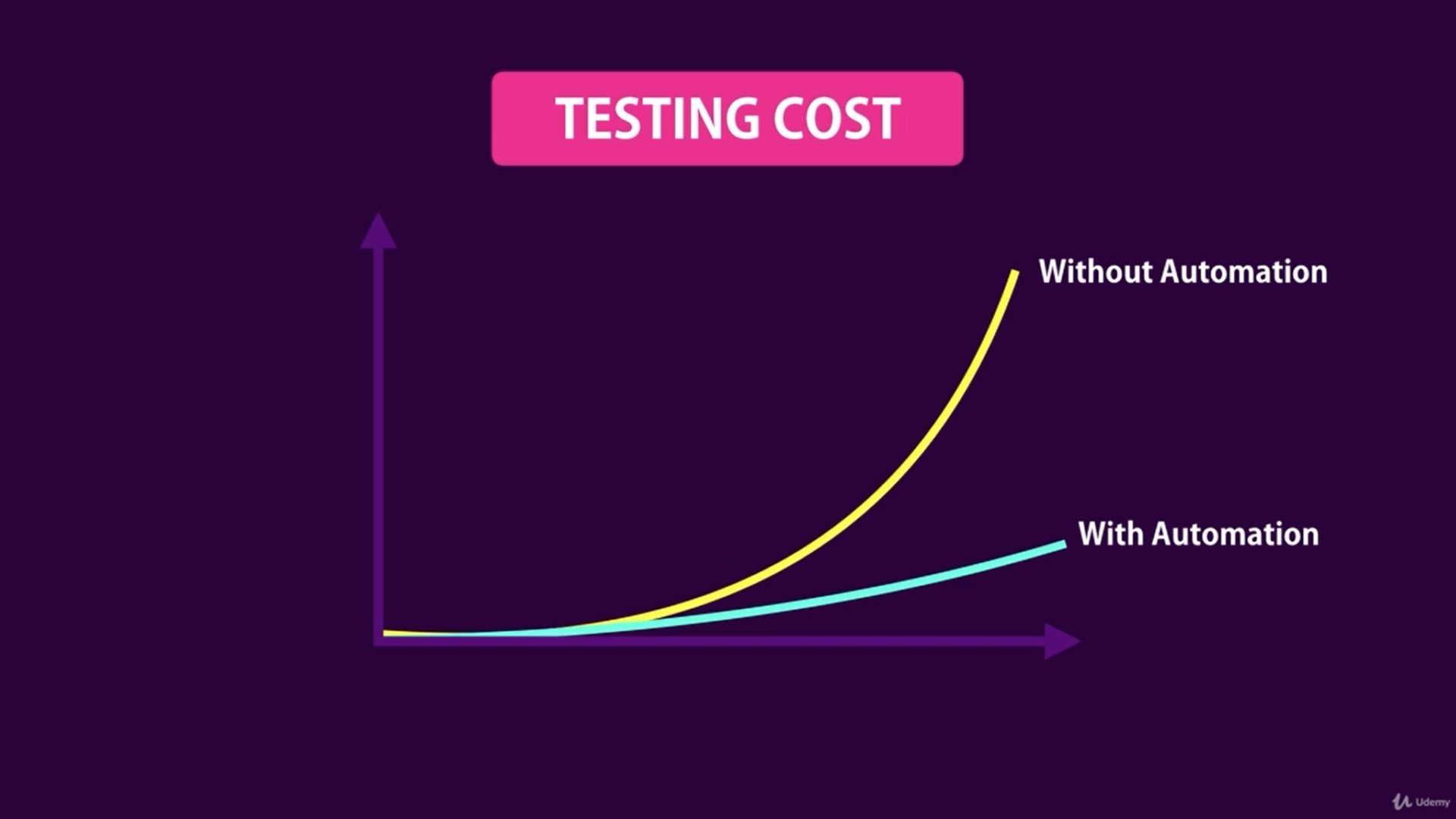


This is a basic calculate function that takes an input and depending on some conditions, it returns different values. If you want to test this function manually, you have to launch your application in the browser, perhaps you have to login or maybe you have to do a few clicks to get to a page to where this function is used.

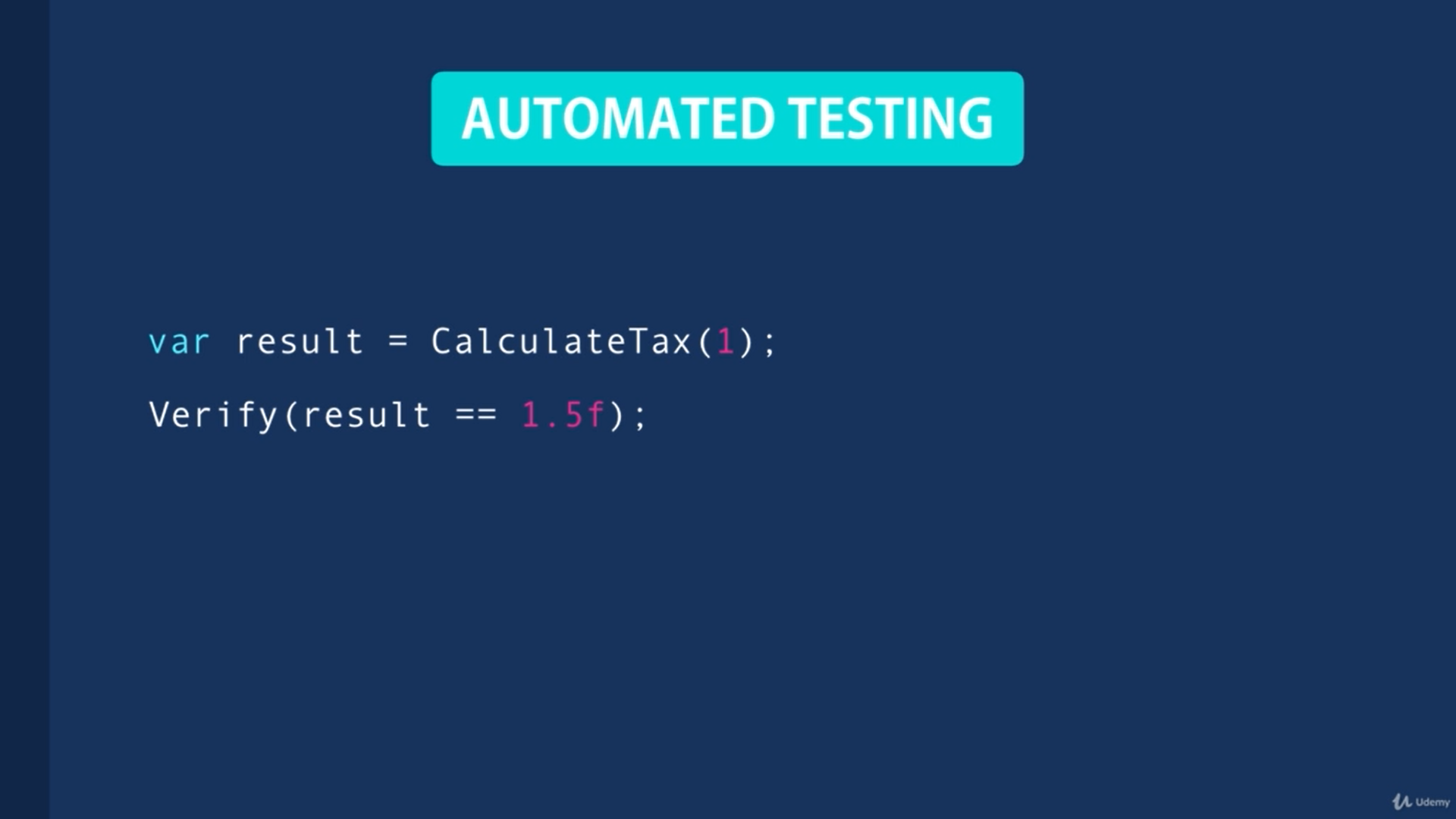
Then you must fill out the form, submit it and see the result of this function on the screen. And then you must repeat using different values in the form. This is very time consuming. In a real application, you have tens or hundreds of functions like this.



As your application grows and complexity, the time required to manually test all the different bits and pieces increases exponentially.



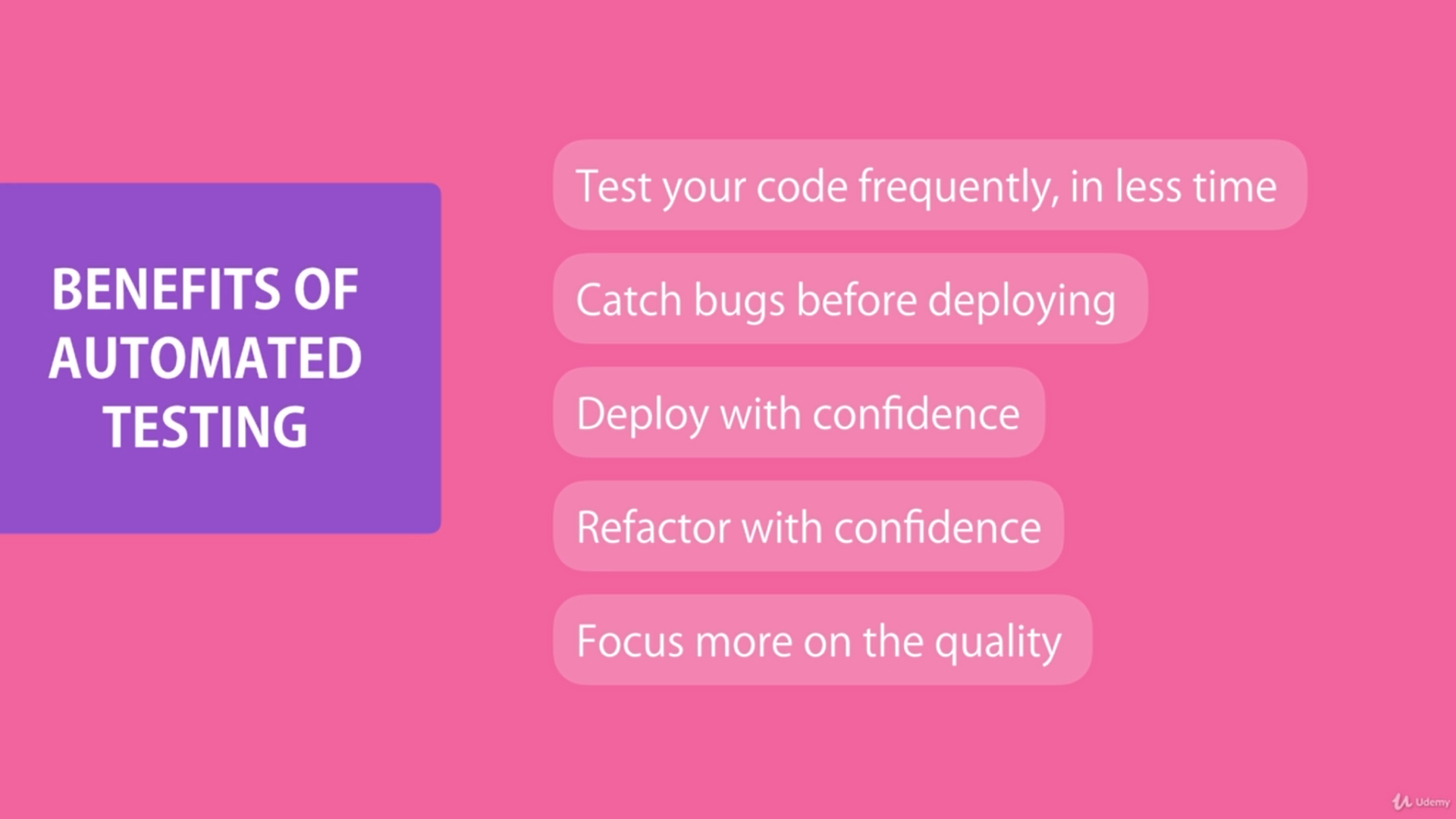
That’s why we use Automation testing. With automation testing, you write code and directly call the function with different inputs and verify that this function returns the right output. Automated tests are repeatable. With this approach, you can test all the execution paths in the function in less than a second.



You can write several hundred or thousands of automated tests for various parts of your application and run them all in just a few seconds.

1. What are the benefits of Automated Testing?

The most important benefit is that you can catch the bugs BEFORE deploying our application. This is extremely important because it allows you to deploy your application with more confidence.



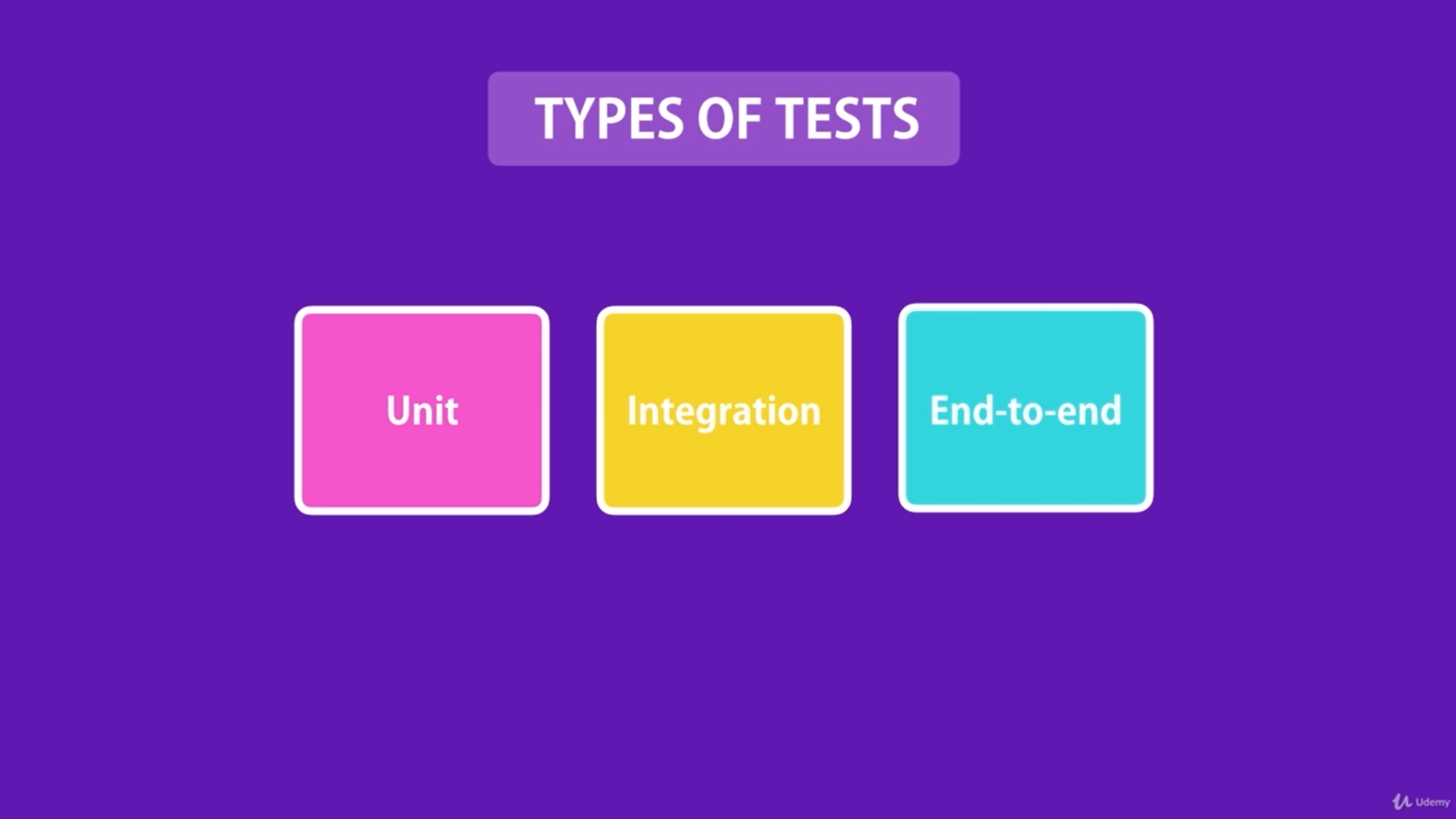
Another benefit of Unit Tests is that they allow you to refactor your code with confidence. Refactoring means changing the structure of your code, without changing its behaviour. If you extract a few lines of a method into a separate private method, that’s refactoring. If you rename a method, that’s refactoring too. You’re changing the structure of your code to make it cleaner and more maintainable, but you’re not changing the functionality.

When you don’t have automated tests, every time you refactor your code you must manually test every part of the application that could be affected by your refactoring. And this is very painful because first, it’s time consuming and secondly as your application grows, you may forget about the parts that need to be tested. With automated tests, every time you refactor your code, you run your tests and make sure you didn’t break anything that used to previously work.

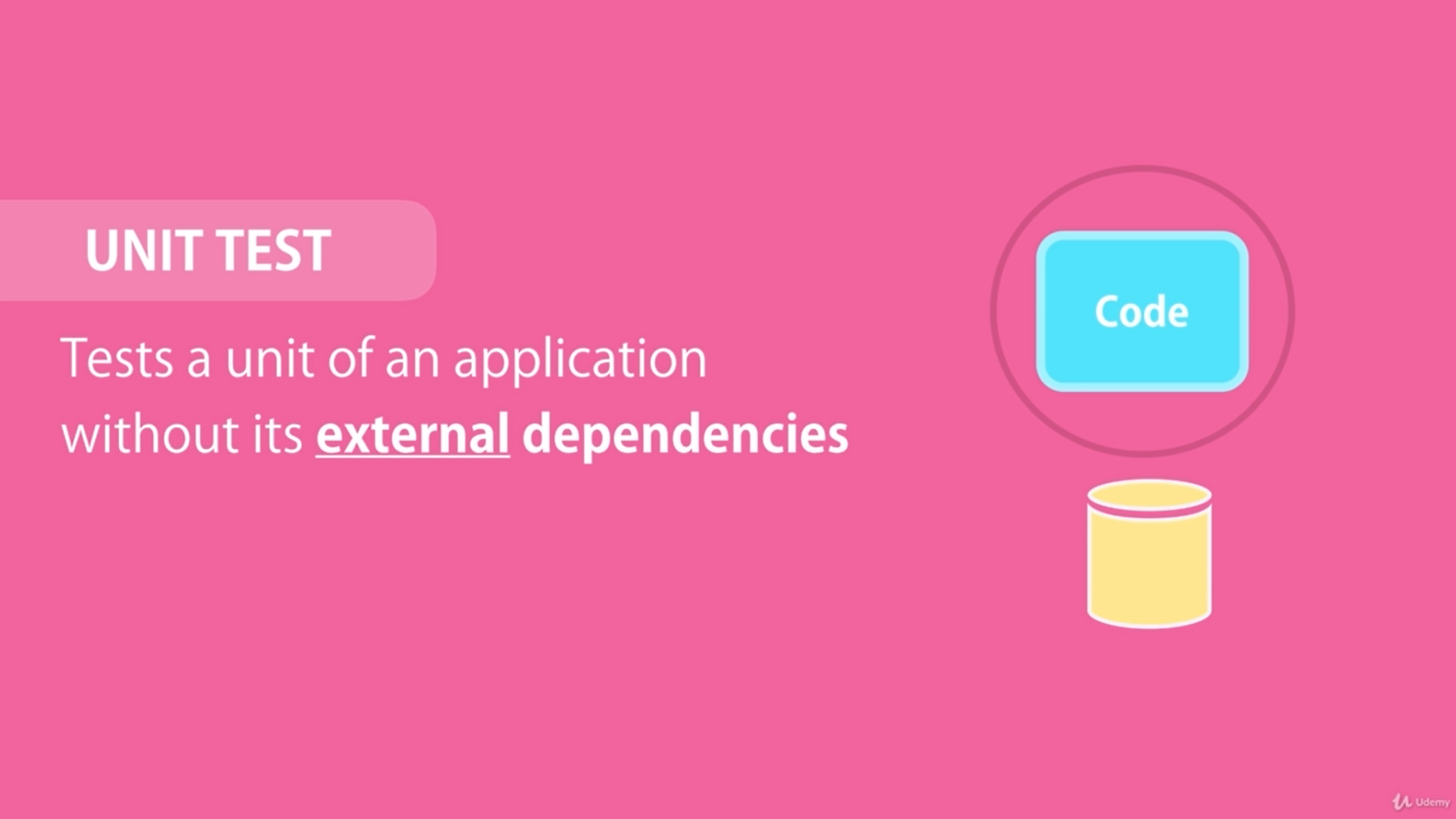
Finally, it helps you focus more on the quality of the methods that you’re writing. You make sure that every method works with different inputs under varying circumstances.

1. What are the different types of tests?

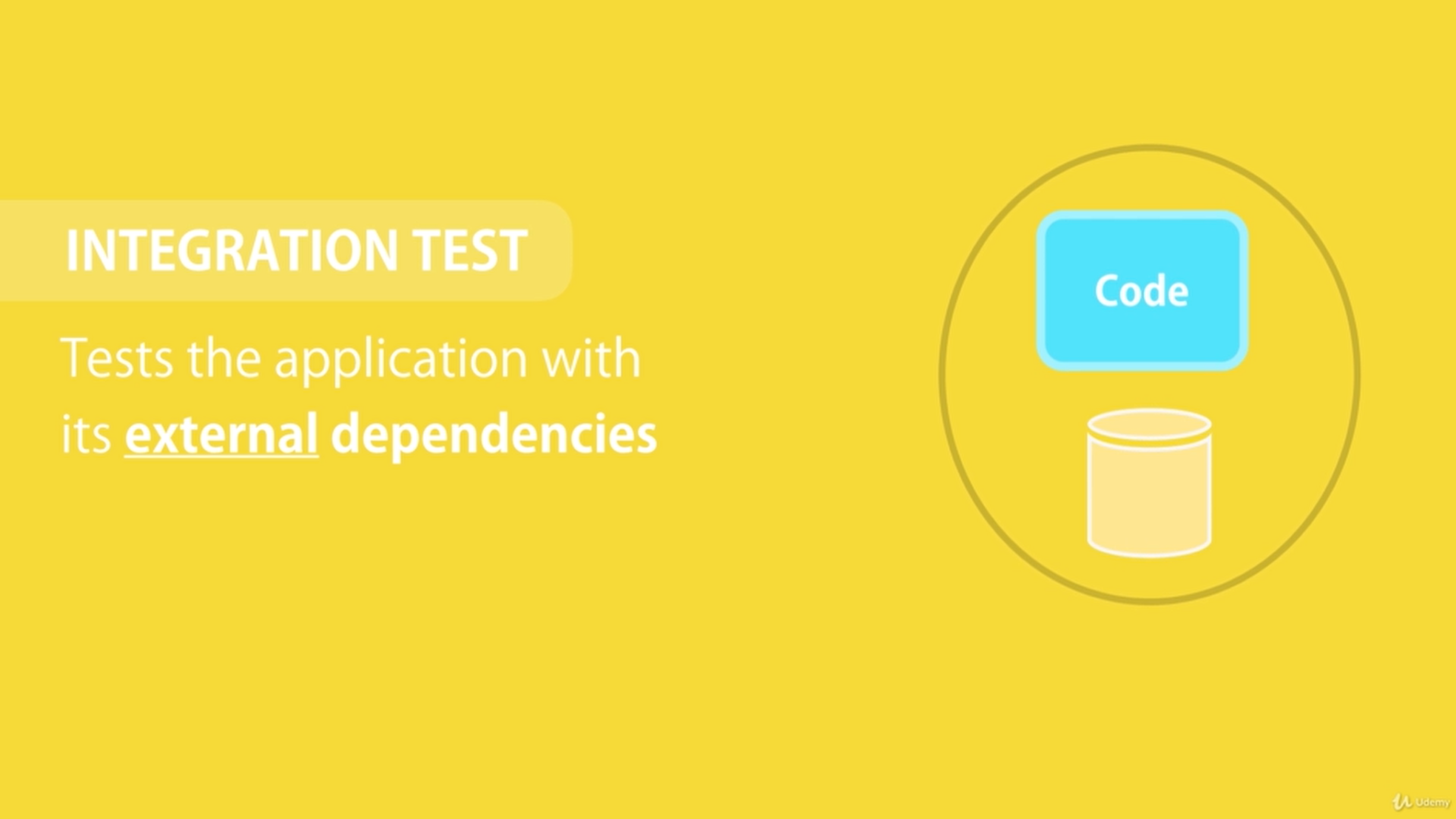
In automated testing, we have 3 types of tests: Unit tests, Integration tests and end-to-end tests.



**Unit Test:** Tests a unit of an application without its external dependencies such as files, databases, message queues, web services and so on. They are Cheap to write, and they execute fast. Since you’re not testing the classes or components with an external dependency you can’t get a lot of confidence in the reliability of your application. That’s when integration test comes to the rescue.



**Integration Test:** Integration testing tests the application with its external dependencies.



These tests take longer to execute because they are often involved in reading and writing to a database. But they give you more confidence in the health of your application.

Traditionally, an integration testing is defined as a test that takes a few units or classes and test their behaviour. Based on this definition, if you test 2 classes together some people believe you’re writing an integration test and not a unit test, even if none of these classes talk to an external resource like database.



Chances are you’ve heard this definition before. This definition is a great recipe for writing fragile tests that are couple to your implementation details. So, as you change the implementation of your classes, these tests are going to break, and you’ll end up wasting a lot of time fixing them. Not only won’t they give you any values, but they slow you down!

**End-To-End test:** Drives an application through its UI. There are specific tools built for creating end-to-end tests. One popular tool is Selenium, which allows us to record the interaction of a user with our application and then play it back and check if the application is returning the right result or not.

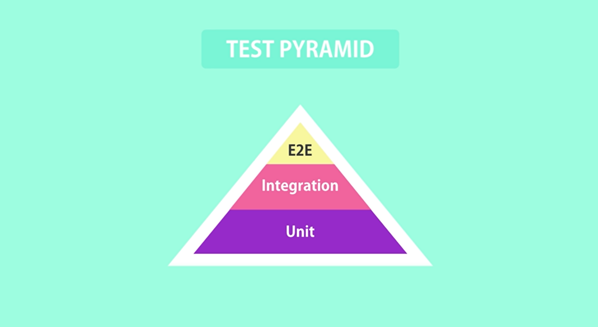
These tests give you the greatest amount of confidence about the health of your application, but they have two big problems:

* The first problem is that they are very slow. Because they require launching the application and testing it through the UI. So, every test is going to launch the application, potentially login, navigate to an internal page, submit a form and inspect the result. Very slow.
* The second problem is that they’re very brittle, because a small enhancement to the application or a small change in the user-interface can easily break these tests.

1. What is Test Pyramid?

Test Pyramid argues that most of your tests should be in the category of Unit Tests, because these tests are easy to write, and they execute quickly. But since they don’t give you much confidence about the health of your application, you should have a bunch of integration tests that test the integration of your application code with its external dependencies. These tests provide many advantages of end-to-end tests, but without the complexities of dealing with the user interface. And finally, you should write very few end-to-end tests for the key functions of the application, but you should not test the edge cases with these end-to-end tests. You only test the happy path and leave the edge cases to unit tests.

The pyramid is just a guideline. It’s not a hard and fast rule you need to follow in every application. The actual ratio between unit, integration and end-to-end tests depends on your project.





1. What are the different testing frameworks?

NUnit: It is one of earliest frameworks.

MSTest: It is Microsoft’s testing framework built into Visual Studio

xUnit: It has gained more popularity over the past couple of years.

All these frameworks give you utility library to write your tests and a test runner which runs your tests and gives you a report of passing and failing tests. Which framework is better?

It depends who you ask and what their definition of better is. Different people love different tools. So, here is my suggestion: Don’t get hung up on the tooling.

**Focus on the fundamentals not the tooling.**