A4Q
Selenium Tester
Foundation
Syllabus

Released
Version 1.3 / 2019

Alliance for Qualification

Revision History

Version	Date	Remarks
Version 1.0	5 August 2018	1. Release version
Version 1.1 / 1.2	2018 - 2019	Minimal corrections
Version 1.3	10 May 2019	Minimal corrections: - Figure 17 / page 31: Correction of double quotes - Section 4.4 / page 77 : Correction of the last bullet points from “test execution layer” to “test definition layer” - Table 2 / page 22: correction of comment tags

Table of Contents

0 Introduction

0.1 Purpose of this Syllabus

This syllabus presents the business outcomes, learning objectives, and concepts underlying the Selenium Tester Foundation training and certification.

0.2 Examinable Learning Objectives and Cognitive Levels of Knowledge

Learning objectives support the business outcomes and are used to create the certified Selenium Tester Foundation exams.

In general, all contents of this syllabus are examinable at a K1 level, except for the Introduction and Appendices. That is, the candidate may be asked to recognize, remember, or recall a keyword or concept mentioned in any of the four chapters. The knowledge levels of the specific learning objectives are shown at the beginning of each chapter, and classified as follows:

- K1: remember
- K2: understand
- K3: apply
- K4: analyze

The definitions of all terms listed as keywords just below chapter headings shall be remembered (K1), even if not explicitly mentioned in the learning objectives.

0.3 The Selenium Tester Foundation Exam

The Selenium Tester Foundation exam will be based on this syllabus and the accredited A4Q training course Selenium Tester Foundation training course. Answers to exam questions may require the use of material based on more than one section of this syllabus and/or the Selenium Tester Foundation training course. All sections of the syllabus and the Selenium Tester Foundation training course are examinable, except for the Introduction and Appendices.
Standards, books, and ISTQB® syllabi may be included as references, but their content is not examinable, beyond what is summarized in this syllabus itself from such standards, books, and ISTQB® syllabi.

The exam shall be comprised of 40 multiple-choice questions. Each correct answer has a value of one point. A score of at least 65% (that is, 26 or more questions answered correctly) is required to pass the exam. The time allowed to take the exam is 60 minutes. If the candidate's native language is not the examination language, the candidate may be allowed an extra 25% (15 minutes) time.

Exams may only be taken after taking the A4Q Selenium Tester Foundation training, since the instructor's evaluation of the candidate's competency in the exercises is part of attaining the certification.

0.4 Accreditation

The A4Q Selenium Tester Foundation training is the only accredited training course.

0.5 Level of Detail

The level of detail in this syllabus allows internationally consistent exams. In order to achieve this goal, the syllabus consists of:

- General instructional objectives describing the intention of the Foundation Level
- A list of terms that students must be able to recall
- Learning objectives for each knowledge area, describing the cognitive learning outcome to be achieved
- A description of the key concepts, including references to sources such as accepted literature or standards

The syllabus content is not a description of the entire knowledge area of Selenium automated testing; it reflects the level of detail to be covered in Foundation Level training courses. It focuses on test concepts and techniques that can apply to all software projects, including Agile projects. This syllabus does not contain any specific learning objectives related to any particular software development lifecycle or method, but it does discuss how these concepts may apply in various software development lifecycles.

0.6 How this Syllabus is Organized

There are four chapters with examinable content. The top-level heading for each chapter specifies the time for the chapter; timing is not provided below chapter level. For the A4Q Selenium Tester Foundation training course, the syllabus requires a minimum of 16.75 hours of instruction, distributed across the four chapters as follows:

Chapter 1: Test Automation Basics 105 minutes

Chapter 2: Internet Technologies for Test Automation of Web Applications 195 minutes

Chapter 3: Using Selenium WebDriver 495 minutes

Chapter 4: Preparing Maintainable Test Scripts 225 minutes

0.7 Business Outcomes

SF-BO-1	Correctly apply test automation principles to build maintainable test automation solution
SF-BO-2	Be able to choose and implement correct test automation tools
SF-BO-3	Be able to implement Selenium WebDriver scripts that execute functional web application tests
SF-BO-4	Be able to implement maintainable scripts

0.8 Acronyms

AKA:	Also Known As
API:	Application Programming Interface
CERN:	European Council for Nuclear Research (French)
CI:	Continuous Integration
CSS:	Cascading Style Sheets
DOM:	Document Object Model
GUI:	Graphical User Interface
HTTP:	HyperText Transfer Protocol
ISTQB®:	International Software Testing Qualifications Board
KDT:	Keyword Driven Testing
REST:	Representational State Transfer
ROI:	Return on Investment
SDLC:	Software Development Life Cycle
SOAP:	Simple Object Access Protocol
SUT:	System Under Test
TAA:	Test Automation Architecture
TAE:	Test Automation Engineer
TAS:	Test Automation Solution
TCP:	Transmission Control Protocol
UI:	User Interface
W3C:	World Wide Web Consortium

Chapter 1 - Test Automation Basics

Keywords

architecture, capture/replay, comparator, exploratory testing, fault attack, framework, hook, pesticide paradox, technical debt, testability, test harness, test oracle, testware

Learning Objectives for Test Automation Basics

STF-1.1	(K2) Explain the objectives, advantages, disadvantages and limitations of test automation
STF-1.2	(K2) Understand the relation between manual and automated tests
STF-1.3	(K2) Identify technical success factors of a test automation project
STF-1.4	(K2) Understand risks and benefits of using Selenium WebDriver
STF-1.5	(K2) Explain the place of Selenium WebDriver in TAA
STF-1.6	(K2) Explain the reason and purpose for metric collection in automation
STF-1.7	(K2) Understand and can compare objectives of using Selenium toolset (WebDriver, Selenium Server, Selenium Grid)

1.1 Test Automation Overview

Test automation is many things. We will limit our discussion of automation in this syllabus to define test automation as the automatic execution of functional tests, designed at least in some ways to simulate a human being executing manual tests. There are many different definitions (see ISTQB® Certified Tester Advanced Level Test Automation Engineer syllabus); this is the one best suited for this syllabus.

While test execution is largely automated, test analysis, test design, and test implementation are usually still manually performed. Creation and deployment of the data used during the testing may be partly automated but is often done manually. Evaluation of the pass/fail status for a test may be part of the automation (via a comparator built into the automation), but not always.

Automation requires the design, creation, and maintenance of different levels of testware, including the environment the tests will be run in, the tools used, the code libraries which supply functionality, the test scripts and test harnesses and the logging and reporting structures to evaluate the test results. Depending on the tools used, monitoring and controlling the execution of the tests may be a combination of manual and automated processes.

There can be many objectives that we try to achieve with functional test automation. Some of these include:

- Improve the efficiency of testing by reducing the cost of each test run
- Test more and other things than we might be able to manually test
- Reducing the amount of time needed for running tests
- Being able to push more testing earlier in the SDLC to find and remove defects from the code earlier (i.e., shift left)
- Increase the frequency with which tests can be run

Like all technology, there are both advantages and disadvantages to using test automation.

Not all advantages are obtainable in all projects, nor do all disadvantages occur in all projects. By strict attention to detail and using good engineering processes, it is possible to increase the good and decrease the bad outcomes that may result from an automation project. One thing is true: an organization has never ***accidentally*** built a successful automation project.

Advantages of automation may include:

- Running automated tests may be more efficient than running them manually
- Performing some tests that cannot be performed at all (or easily) manually (such as reliability or efficiency tests).
- Reducing the time needed for test execution, allowing us to run more tests per build
- Increasing the frequency that some tests can be run
- Freeing up manual testers to run more interesting and complex manual tests (e.g., exploratory tests)
- Reducing mistakes made by bored or distracted manual testers, especially when repeating regression tests
- Executed tests earlier in the process (e.g., placed on the continuous build machine to run unit, component, and integration tests automatically), providing quicker feedback on system quality and removing defects from the software earlier
- Executing tests outside of normal business hours
- Increasing confidence in the build

Disadvantages can include:

- Increases in costs (including high startup costs)
- Delays, costs, and mistakes associated with testers as they learn new technologies
- In worst cases, complexity may become overwhelming
- Unacceptable growth in the size of the automated tests, possibly exceeding the size of the system under test (SUT)
- Software development skills needed on test team or providing a service to the test team
- Considerable maintenance of tools, environments, and test assets required
- Technical debt is easy to add, especially when extra programming is added to improve context and reasonableness of the automated tests, but difficult to reduce (as with all software)

- Automation requires all the processes and disciplines of software development
- Concentrating on automation can make testers lose sight of risk management for the project
- The pesticide paradox is increased when automation is used since exactly the same test is run each time
- False positives occur when automation failures are not SUT failures, but are due to defects in the automation itself
- Without clever programming in the automated tests, tools are literal minded and stupid; testers are not
- Tools tend to be single threaded – that is, they are only looking for one outcome to an event—humans can figure out what happened and determine on the fly if it was correct

There are a lot of constraints on an automation project; some of these can be mitigated by clever programming and good engineering practices, some cannot. Some of these limitations are technical, some are managerial. These limitations include:

- Unrealistic expectations by management which often drives the automation in the wrong direction
- Short term thinking which can destroy an automation project; only by thinking long term can the automation program succeed
- Organizational maturity is required to succeed; automation based on poor testing processes only delivers bad testing faster (sometimes even slower)
- Some testers are perfectly happy with manual testing and do not want to automate
- Automated test oracles may be different than manual test oracles, requiring that they be identified
- Not all tests can or should be automated
- Manual testing will still be required (exploratory testing, certain fault attacks, etc.)
- Test analysis, design, and implementation are still likely to be manual
- Human beings find most bugs; automation can only find what it is programmed to find and is limited by the pesticide paradox
- False sense of security due to large numbers of automated tests running without finding many bugs
- Technical problems for the project when using bleeding edge tools or techniques
- Requires cooperation with development, which can create organizational issues

Automation can and often does succeed. But that success is only the result of attention to detail, good engineering practices, and very hard work over the long term.

1.2 Manual vs. Automated Tests

Early versions of test automation tools were an abysmal failure. They sold really well, but rarely worked as advertised. Part of the reason is that they did not model software testing at all well, nor did they understand the role of the tester in the testing process.

The basic record/playback (aka capture/replay) tool was sold with some version of the following instructions:

> ***Connect the tool to your system to test. Turn the switch on to start recording. Have the tester perform the test. After completing, switch off the tool. It will generate a script (in a programming language) which will do exactly what the tester did. You can play that script every time you want to run the test.***

Often these scripts failed to work even the first time they were run. Any change in screen context, timing, GUI properties, or hundreds of other things would make the recorded script fail.

To understand test automation, you must understand what a manual test script is and how it is used.

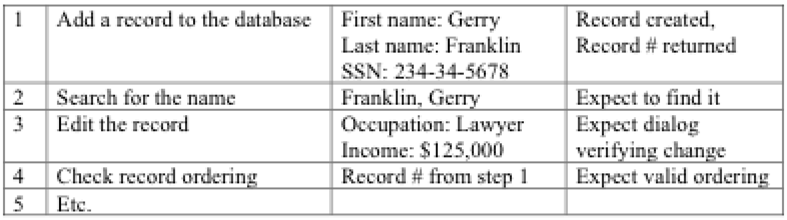
At its bare minimum, a manual test script tends to have information in three columns.

The first column will contain an abstract task to do. Abstract, so it does not need to be changed when the actual software changes. For example, the task “Add Record to Database” is an abstract idea. No matter what version of what database, this abstract task can be performed—assuming a manual tester has the domain knowledge to translate it into concrete actions.

The second column tells the tester what data to use to perform the task.

The third column tells the tester what behavior to expect.

Table 1: Manual Test Fragment



1	Add a record to the database	First name: Gerry Last name: Franklin SSN: 234-34-5678	Record created, Record # returned
2	Search for the name	Franklin, Gerry	Expect to find it
3	Edit the record	Occupation: Lawyer Income: $125,000	Expect dialog verifying change
4	Check record ordering	Record # from step 1	Expect valid ordering
5	Etc.		

Manual testing has worked very well for a lot of years because we have been able to create these manual test scripts. However, the script by itself is not what allows the testing to be done. It is only when the script is in the hands of a knowledgeable manual tester that we can extract the value.

What does the tester add to the script to allow them to run the test correctly? **Context** and **reasonableness**. Each task is filtered through the knowledge of the tester: “What database? What table? How do I perform this task with this version of this database?” Context leads to some of the answers, reasonableness leads to others to allow the tester to succeed at the tasks in the test script.

An automated test run through an automated tool has limited context and reasonableness. To perform a particular task (e.g., “Add Record”) the tool must be sitting in the exact location to perform the task. The assumption is that the last step of the automated test case put them in the exact location where they can perform “Add Record”. And if it didn’t? Too bad! Not only will the automation fail, but the actual error message is likely to be meaningless, since the real failure was probably during the previous step. A tester running the test will never have this issue.

Likewise, when running a manual test, the result of an action is checked by the manual tester. For example, if the test required the tester to open a file (i.e., task = “Open File”), there are a variety of things that can happen. The file might open. An error message, warning message, informational message, timing message, and about a dozen other things can result. In each case, the manual tester simply reads the message, applies reasonableness and context to the issue, and does the right thing.

That is the crux of the difference between an automated script and a manual script. The manual test script only adds value in the hands of a manual tester. We could add lines in the manual script to help the tester understand how to deal with an issue, but mostly we don’t because the manual tester has a human brain, good intelligence and experience, allowing them to figure out the issue on their own. And, in a worst-case scenario, they can pick up their phone and ask an expert.

Consider some of the questions that the manual tester can answer:

- How long must I wait for something to happen?
- What happens if I get a fixable return result? (e.g., tried to open a file but got a “Drive not mapped” error message)
- What happens if I get a warning?
- What happens if I am supposed to wait 5 seconds, but it actually took 5.1 seconds?

None of those results can be natively handled by an automation tool by itself. An automation tool has no intelligence; it is just a tool. An automated script, if recorded, has limited understanding of what it can do if something other than the expected occurs. One size fits all: throw an error.

However, an automated script may have intelligence built into it by the automator via programming. Once automators understood that intelligence can be added to the automation script, we were able to start making automation work better. By programming, we can capture the thought processes that a manual tester has, context and reasonableness, and start making automation add more value by more often actually completing the test rather than failing it early. Notice, however, that there is no such thing as a free lunch. Extra programming may also cause more maintenance to be needed later on.

Not all manual tests should be automated. Because an automated script requires more analysis, more design, more engineering and more maintenance than a manual script, we must figure in the cost of creating it. And, once we create the automated script, forever more we will have to maintain it, and that must be considered also. When the SUT changes, often the automated scripts will need to be changed concurrently.

Inevitably, we will find new ways that an automated script might fail: simply getting a return value we never saw before will make the automation fail. When that happens, we will need to change, re-test, and re-deploy the script. These are generally not problems with manual testing.

A business case needs to be made before—and while—we automate our testing. Will the overall cost of automating this test be returned by being able to run it N times over the next M months? Many times, we can determine the answer to be no. Some manual tests will remain because there is no positive return on investment (ROI) for automating them.

Some manual tests simply cannot be automated because the thought process of the manual tester is essential to the success of the test. Exploratory testing, fault attacks, and some other types of manual testing are still required for success of a testing effort.

1.3 Success Factors

Automation does not succeed by accident. Success usually requires:

- A long-term plan aligned with the needs of the business
- Solid, intelligent management
- Stark attention to detail
- Process maturity
- An architecture and framework which are formalized
- Proper training
- Mature levels of documentation.

The ability to automate often rests on the testability of the system under test (SUT). There are many times that the interfaces of the SUT are just not suitable for testing. Getting special or private interfaces (often called hooks) from the developers of the system can often mean the difference between succeeding or failing at automation.

There are several different levels of interface that we can automate an SUT at:

- The GUI level (often the most brittle and failure prone level)
- The API level (using application programming interfaces that the developers make available for public or protected use)
- Private hooks (APIs that are specifically for testing)
- The protocol level (HTTP, TCP, etc.)
- The service level (SOAP, REST, etc.)

Note that Selenium works at the UI level. This syllabus will contain tips and techniques to reduce the brittleness and enhance the usability while testing at this level.

The following are all success factors for automation:

- Management which has been educated to understand what is and is not possible (unrealistic expectations by management is one of the biggest reasons for software test automation projects to fail)
- A development team for the SUT(s) that understands automation and is willing to work with the testers when needed
- SUT(s) that are designed for testability
- Developing a business case for the short, medium and long term
- Having the right tools to work in the environment and with the SUT(s)
- The right training, including both test and development disciplines
- Having a well-designed—and well documented—architecture and framework to support the individual scripts (the Advanced Test Automation Engineer syllabus calls this the TAS (test automation solution))
- Having a well-documented test automation strategy that is funded and supported by management
- A formal and well-documented plan for maintenance of the automation
- Automating at the right interface level for the context of tests necessary

1.4 Risks and Benefits of Selenium WebDriver

WebDriver is a programming interface for developing advanced Selenium scripts using the following programming languages:

- C#
- Haskell
- Java
- JavaScript
- Objective C
- Perl
- PHP
- Python
- R
- Ruby

Many of these languages also have open source testing frameworks available.
Browsers supported by Selenium (and the component needed to test with) include:

- Chrome (chromedriver.exe)
- Internet Explorer (IEDriverServer.exe)
- Edge (MicrosoftWebDriver.msi)
- Firefox (geckodriver.exe)
- Safari (safaridriver)

- HtmlUnit (HtmlUnit driver)

Selenium WebDriver works by using its APIs (Application Programming Interfaces) to make direct calls to a browser using each different browser's native support for automation. Each supported browser works slightly differently.

As with any tool, there are both benefits and risks to using Selenium WebDriver.
Many of the benefits that an organization may profit from are the same as any other automation tool, including:

- Test execution can be consistent and repeatable
- Well suited for regression testing
- Because it tests at the UI level, it may catch defects missed by testing at API level
- Lower up-front investment because it is open source
- Works with different browsers so compatibility testing is possible
- Supports different programming languages so can be used by more people
- Because it requires deeper understanding of the code, useful on Agile teams

With benefits, go risks. The following are risks that come with the use of Selenium WebDriver.

- Organizations often get so wrapped up in GUI testing that they forget that the testing pyramid suggests more unit/component testing be done
- When testing for a CI (continuous integration) workflow, this automation may make the build take much longer to complete than desirable
- Changes to the UI cause more damage to browser level tests than they do to unit or API-level tests
- Manual testers are more efficient at finding bugs than is automation
- Tests that are difficult to automate might get skipped
- Automation needs to be run often to return a positive ROI (return on investment.) If the web application is fairly stable, the automation may not pay for itself.

1.5 Selenium WebDriver in Test Automation Architecture

The TAA (Test Automation Architecture) as defined by ISTQB® in its Certified Tester Advanced Level Test Automation Engineer (TAE) syllabus is a set of layers, services, and interfaces of a TAS (Test Automation Solution).

The TAA consists of four layers (see illustration below):

- Test generation layer: supports manual or automated design of test cases
- Test definition layer: supports the definition and implementation of test cases and /or suites

- Test execution layer: supports both the execution of automated tests and the logging/reporting on the results
- Test adaptation layer: provides the necessary objects and code to interface with the SUT (system under test) at various levels.

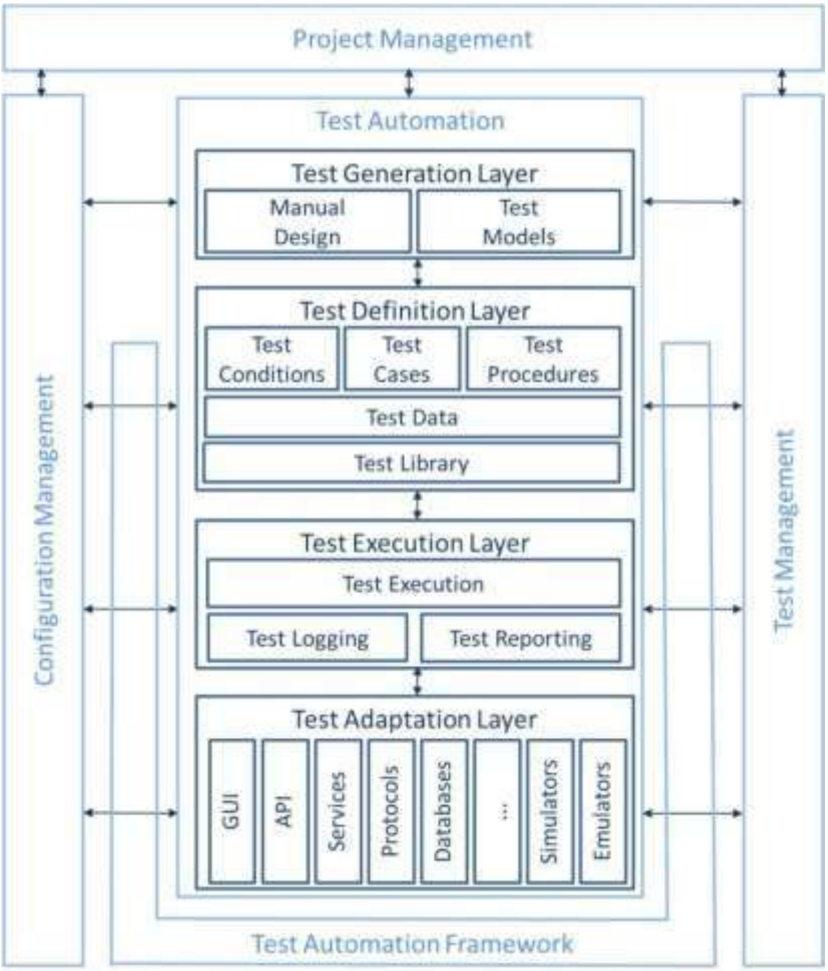


Figure 1: A generic TAA (from TAE syllabus)

Selenium WebDriver fits into the test adaptation layer, providing a programmatic way to access the SUT through the browser interface.

The test adaptation layer facilitates a separation of the SUT (and its interfaces) from the tests and test suites we want to run against the SUT.

Ideally, this separation allows the test cases to be more abstract and disconnected from the system that is being tested. When the system under test changes, the test cases themselves may not need to change, assuming that the same functionality is being provided—just in a slightly different way.

When the SUT changes, the test adaptation layer, in this case, WebDriver, allows modification of the automated test, allowing it to run the actual concrete test case against the modified interface of the SUT.

The automated script using WebDriver effectively uses the API to communicate between the test and the SUT as follows:

- The test case calls for a task to executed
- The script calls an API in WebDriver
- The API connects to the appropriate object in the SUT
- The SUT responds (hopefully) as requested
- The success or failure is communicated back to the script
- If successful, the next step in the test case is called in the script

1.6 Purpose for Metrics Collection in Automation

There is an old saying in management, “What gets measured gets done.” Measurements are one of those things that many automators like to ignore or obfuscate, because they are often difficult to quantify and prove.

The problem is that automation tends to be quite expensive in both human effort and tool resources. There is little to no positive return on investment in the short run; value comes over the long run. Years rather than months. Asking management for a large investment in time and resources and not making a provable business case for it (including the collection of metrics to provide proof) is asking them to believe in us, to have faith. Unfortunately, to upper management, those phrases might as well be obscenities.

We should collect meaningful measurements where we can that show the value of the automation if we want the automation project to survive the bean counters. The Advanced Test Automation Engineer (TAE) syllabus mentions several areas where metrics would be useful, including:

- Usability
- Maintainability
- Performance
- Reliability

Many of the metrics that are mentioned in the TAE syllabus are likely to be more useful for large scale automation teams rather than small automation projects as might use Selenium WebDriver. We are going to concentrate on smaller projects in this section. If you are working on a large, complex automation project, please refer to TAE syllabus.

Collecting inappropriate metrics is likely to be a distraction to a small team just trying to get a project off the ground for the first time. One problem with identifying meaningful metrics to collect is that, while test automation execution time tends to be smaller than the equivalent manual test time, the analysis, design, development, troubleshooting and maintenance of automated tests tends to take much longer than the same work for manual tests.

Trying to capture meaningful metrics to determine the business case (ROI) often becomes a case of comparing apples to oranges. For example, it might take 1 hour to create a specific manual test case. Thirty minutes to run it. Assume we plan on running the test three times for this release. We need 2.5 hours for that test. Simple measurements for manual testing.

How much will it cost to automate that same test, however? First thing is to determine if there is any value in automating that test at all. Then, assuming we decide it should be automated, figuring how to automate it, writing the code needed, debugging the code and ensuring that it is doing what we expect might take several hours. Every test is liable to have different challenges; how do we estimate that?

The manual test, being abstract, likely does not need to be changed when the GUI of the SUT changes. The automated script likely would need to be modified for even minor changes to the GUI. We can minimize that by good architecture and framework
designs, but changes will still take time. So how do we estimate how many expected changes there will be?

When there is a failure for a manual test, troubleshooting it is usually straightforward. The test analyst can usually analyze what went wrong easily so they can write the incident report.

An automation failure, however, might take more time to troubleshoot (see logging section below).

How do you calculate how many times the test might fail—especially when those failures are often due to the automation itself, not a failure of the SUT? Just to make it harder to calculate the return on investment, the automation requires a great deal of investment up front that is not required for manual testing. This investment includes purchase of one or more tools, suitable environments to run the tools in, creation of the architecture and framework to facilitate running the automation, training (or hiring) automators, and other fixed costs.

That investment needs to be added into any ROI calculation that we might make. Because those are fixed costs, achieving positive ROI likely means that we need to automate and run large numbers of tests. That requires a higher level of scalability which itself will make the automation cost more.

Another problem with many automation metrics is that they often must be estimated rather than measured directly; this often means that they get twisted in an effort to prove that the automation is worthwhile.

For a small, startup project, here are some metrics that might be useful:

- Fixed costs to get the automation up and running
- Regression test effort that has been saved by the automation
- Effort expended by automation team supporting the automation
- Coverage
  - At unit test level, via statement/decision coverage
  - At integration test level, interface or data flow coverage
  - At system test level, requirement, feature, or identified risk coverage
  - Configurations tested
  - User stories covered (in Agile)
  - Use cases covered
- Covered configurations tested
- Number of successful runs between failures
- Patterns of automation failures (looking for commonality of problems by tracking root causes of failures)
- Number of automation failures found as compared to SUT failures found by automation

Here is the final point about metrics. Try to find meaningful metrics to collect about your project that help explain why the project is valid and important. Negotiate with management, making sure they understand that proving the value of automation is difficult at the start and often takes a lot of time to make reasonable progress towards meaningful metrics. It is tempting to use rosy scenarios and creative imagination to prove the value of the automation. Getting caught at that by management, however, could be the kiss of death to the project.

When done correctly, an automation project has a good chance of adding value to the organization, even if it is difficult to prove.

1.7 The Selenium Toolset

The Selenium WebDriver is not the only tool from the Selenium stable. The open-source Selenium ecosystem consist of four tools, each of which play different roles in test automation:

- Selenium IDE
- Selenium WebDriver
- Selenium Grid
- Selenium Standalone Server

During the long history of Selenium tools, there existed the tool called Selenium RC, which implemented Selenium version 1. This tool is no longer used.

Selenium IDE is an add-on to Chrome and Firefox web browsers. Selenium IDE does not work as standalone application. Its main function is recording and playing back user actions on web pages.

Selenium IDE also allows an automator to insert verification points during recording. The recorded scripts can be saved to disk as HTML tables or exported to several different programming languages.

The main advantages of Selenium IDE are its simplicity and reasonably good element locators. Its main disadvantage is lack of variables, procedures and control flow instructions; as such, it is really not useful for creating robust automated tests. Selenium IDE is mainly used for recording provisional scripts (e.g., for debugging purposes) or just for the sake of finding locators.

Selenium WebDriver is mainly a framework allowing test scripts to control web browsers. It is based on HTTP and has been standardized by W3C. This syllabus shows basic features of this protocol in **Fehler! Verweisquelle konnte nicht gefunden werden.**. Selenium WebDriver has bindings for many different programming languages e.g. Java, Python, Ruby, C#. Throughout this syllabus Python is used to show examples of how to use different WebDriver objects and their methods.

Using libraries which implement the WebDriver API for diverse programming languages enables test automators to combine WebDriver’s ability to control web browsers with the power of general programming languages. This permits automators to utilize other languages’ libraries to build complex test automation frameworks featuring logging, assertion handling, multi-threading, and much more.

To achieve long-term success, building test automation frameworks has to be done with care and good design principles as described in section **Fehler! Verweisquelle konnte nicht gefunden werden.**.

Some web browsers need additional WebDriver processes to start new test instances and to control them. In that case a script calls commands from the library and the library through WebDriver process sends commands to the web browser. This will be discussed in chapter three.

Another tool which can be useful in a test environment is Selenium Grid. It enables running test scripts across multiple machines with different configurations. It allows distributed and simultaneous execution of test cases. The architecture of Selenium Grid is very flexible. It can be configured to use many physical or virtual machines with different combinations of operating systems and versions of web browsers.

In the center of Selenium Grid, there is a hub that controls other nodes and acts as a single point of contact for test scripts. Test scripts which execute WebDriver commands do not need (in most cases) to be changed to work on different operating systems or web browsers.

The last tool from Selenium ecosystem the we mention in this syllabus is Selenium Standalone Server. This tool is written in Java and is delivered as a jar file that implements hubs and nodes functions for Selenium Grid. This tool needs to be started separately (outside of test scripts) and configured properly to play its role in test environment.

A more detailed description of Selenium Grid and Selenium Standalone Server is beyond the scope of this syllabus.

Chapter 2 - Internet Technologies for Test Automation of Web Applications

Keywords

CSS selector, HTML, tag, XML, XPath

Learning Objectives for Internet Technologies for Test Automation of Web Applications

STF-2.1	(K3) Understand and can write HTML and XML documents
STF-2.2	(K3) Apply XPath to search XML documents
STF-2.3	(K3) Apply CSS locators to find elements of HTML documents

2.1 Understanding HTML and XML

2.1.1 Understanding HTML

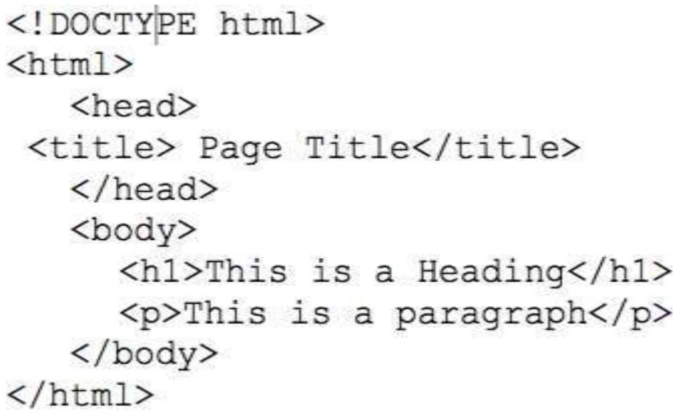
It would not be an exaggeration to say that HTML (HyperText Markup Language) opened up the World Wide Web. An HTML document is a plain text file which contains elements which specify certain contextual meanings when the document is parsed. The elements work together to identify how a browser should render those parts of the document. Essentially, HTML describes the structure of a web page semantically.

Perhaps the most important benefit to using HTML to specify web pages is the universal applicability of the language. When written appropriately, any browser on any computer system can render the page correctly.

What the page actually looks like may change in some ways, depending on the type of computer (e.g., PC versus a smart phone), the monitor, the internet connection speed, and the browser.

Credit for inventing the World Wide Web is given to Timothy Berners Lee. While working for CERN (European Council for Nuclear Research) in 1980, he proposed using hypertext to enable sharing and updating information among researchers. Then, in 1989, he implemented the first effective communication between an HTTP (HyperText Transfer Protocol) client and server, ushering in the World Wide Web and changing the world as we know it.

HTML elements are introduced and often surrounded by tags which are defined by angle brackets as seen below:



```
<!DOCTYPE html>
<html>
   <head>
 <title> Page Title</title>
   </head>
   <body>
     <h1>This is a Heading</h1>
     <p>This is a paragraph</p>
   </body>
</html>
```

Figure 2: HTML Sample

Some tags directly introduce content into the page being rendered (e.g., **<img …./>** will result in an image being placed in the page.) Other tags surround and provide semantic information about how the element is to be rendered (as seen above for the **<h1>….</h1>** heading element. We will discuss tags below.

Throughout most of the history of the World Wide Web, the version of HTML that was used was stable at 4.01. However, in 2014, the governing body which controls HTML released the current version, HTML 5.

HTML is somewhat flexible, allowing some variation in the way tags are used (e.g., some tags may not have a closing tag.) This has caused some browsers to render pages erratically. XML, which will be discussed below, is a language which is more restrictive than HTML, requiring that each page be “well formed” with every opening tag being balanced with a closing tag. When written in a well-formed way (i.e., all open tags are matched with close tags), HTML is a subset of XML.

Automation with Selenium requires an understanding of HTML tags. To automate any GUI, the automator must be able to identify each unique control in the screen being manipulated. Searching through the HTML page allows the automator to distinctively detect the controls that will be placed on the browser-rendered page. To find the controls, the automator must understand the layout and logic of the HTML page; that is done by understanding and parsing through the tags.

HTML elements usually have a start tag and an end tag. The end tag is the same as the start tag except the end tag is preceded by a slash as shown below:

<p>Paragraph text**</p>**

Some elements can be closed within the open tag; for example, the empty line break element as follows:

<br />

A less strict implementation of the line break, which might cause problems for some browsers would be:

<br>

Following are tags that every Selenium automator should understand.

Table 2: Basic HTML Tags

Tag	**Used for**
<html> … </html>	**Signifies root of HTML document**
<!DOCTYPE>	**Defines document type (not needed in HTML 5)**
<head> … </head>	**Definition and meta data for document**
<body> … </body>	**Defines main content for the document**
<p> … </p>	**Defines a paragraph**
<br />	**Inserts a single line break**
<div> … </div>	**Defines a section in the document**
<!-- … -->	**Defines a comment (may be multi-line)**

Heading tags define different levels of headers. The actual format of the text (size, boldness, font) can be specified in CSS stylesheets.

Table 3: Heading Tags

Tags	**Renders:**
<h1>Heading 1 </h1>	**Heading 1**
<h2>Heading 2 </h2>	Heading 2
<h3>Heading 3 </h3>	Heading 3
<h4>Heading 4 </h4>	Heading 4
<h5>Heading 5 </h5>	Heading 5
<h6>Heading 6 </h6>	Heading 6

Links and images are fundamental to well-designed browser pages and are easy to create with HTML.

<a href="*URL***">***link text***</a>**

This combination of symbols starts with an **<a …>** anchor tag and ends with an **</a>**. These define a hyperlink which can be clicked on. The **href="***URL***"** is an attribute which signifies the destination of the link. The *link text* between the tags represents the text that will appear in the link to be clicked on to take the user to the URL target.

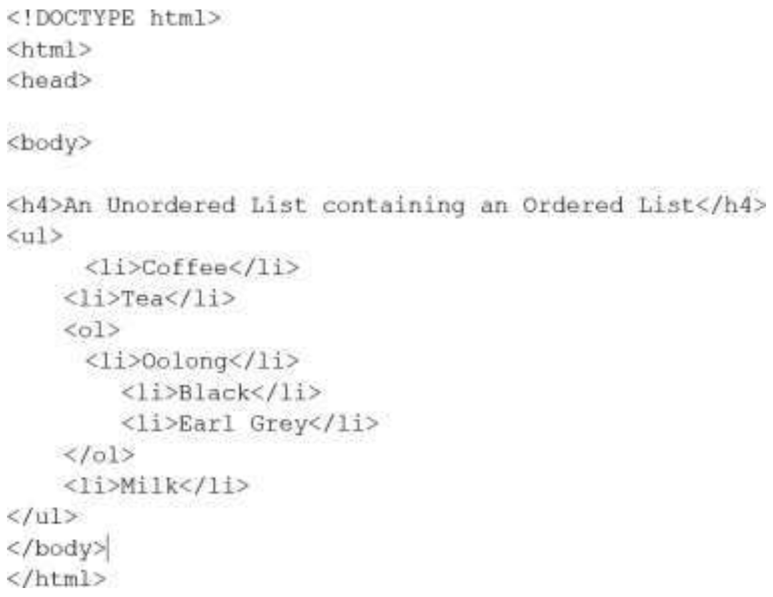
<img src="*pulpitrock.jpg***" alt="***Mountain View***" />**

The basic tag here, **<img…/>**, defines an image that will be placed at this point in the document. The attribute **src="***pulpitrock.jpg***"** is the link address of the actual image that will be shown. The other attribute, **alt="***Mountain view***"**, represents the text that will be shown if the image cannot be found or displayed.

Table 4: List and Table Tags

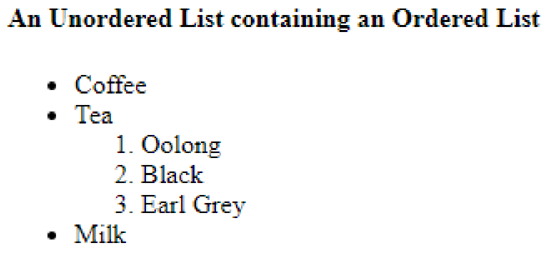
Tag	Used for
<ul> … </ul>	Defines an unordered (bulleted) list
<ol> … </ol>	Defines an ordered (numbered) list
<li> … </li>	Defines a list item (for <ul> or <ol>)
<table> … </table>	Defines an HTML table
<tr> … </tr>	Defines a table row
<th> … </th>	Defines the column header for a table
<td> … </td>	Defines a table data cell
<tbody> … </tbody>	Groups body content in an HTML table
<thead> … </thead>	Defines an HTML table header
<tfoot> … </tfoot>	Defines an HTML table footer
<colgroup> … </colgroup>	Groups table columns for formatting

Lists are simple to define and render. The following HTML code will render the list that follows it.



```
<!DOCTYPE html>
<html>
<head>

<body>

<h4>An Unordered List containing an Ordered List</h4>
<ul>
     <li>Coffee</li>
    <li>Tea</li>
    <ol>
     <li>Oolong</li>
        <li>Black</li>
        <li>Earl Grey</li>
    </ol>
    <li>Milk</li>
</ul>
</body>
</html>
```

Figure 3: HTML List Example



An Unordered List containing an Ordered List

- Coffee
- Tea
    1. Oolong
    2. Black
    3. Earl Grey
- Milk

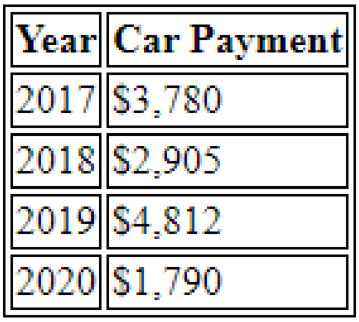
Figure 4: An Unordered List Containing an Ordered List

Tables are also straight-forward to create and render. Test resultant data is often returned in tables so automators often work with them. The following code will render the table as shown below the code.

```
<!DOCTYPE html>
<html>
     <head>
          <style>
               table, th, td {
                   border: 1px solid black;
               }
          </style>
     </head>
     <body>

     <table>
          <tr>
               <th>Year</th>
               <th>Car Payment</th>
          </tr>
          <tr>
               <td>2017</td>
               <td>$3,780</td>
          </tr>
          <tr>
               <td>2018</td>
               <td>$2,905</td>
          </tr>
          <tr>
               <td>2019</td>
               <td>$4,812</td>
          </tr>
          <tr>
               <td>2020</td>
               <td>$1,790</td>
          </tr>
     </table>
     </body>
</htm
```

Figure 5: Table Generation Code

Table 5: Rendered Table



Year	Car Payment
2017	$3,780
2018	$2,905
2019	$4,812
2020	$1,790

HTML forms and the associated controls are used to gather input from users. Following are the tags needed to render the forms and the controls on them. Selenium WebDriver users often have to interact with these forms and controls to automate their tests.

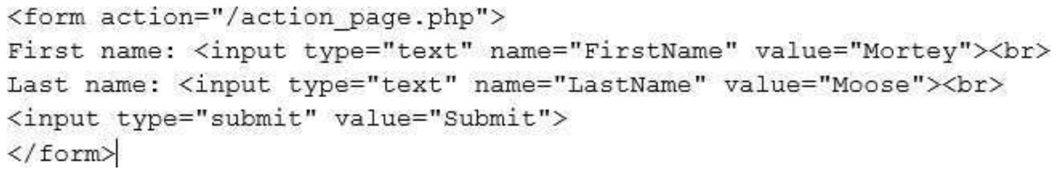
The following tags are used to create controls on the screen.

<form> … </form>

Defines an HTML form for user input.

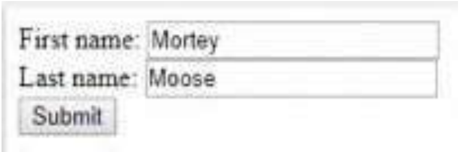
<input>

Defines an input control. The type of control is defined by the attribute **type=**. Possible types include text, radio, checkbox, submit, etc. For example, the following lines will render as follows:



```
<form action="/action_page.php">
First name: <input type="text" name="FirstName" value="Mortey"><br>
Last name: <input type="text" name="LastName" value="Moose"><br>
<input type="submit" value="Submit">
</form>
```

Figure 6: Form Listing



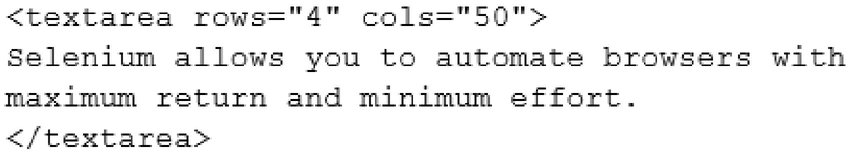
First name: Mortey

Last name: Moose

Figure 7: Input Fields from Form Listing

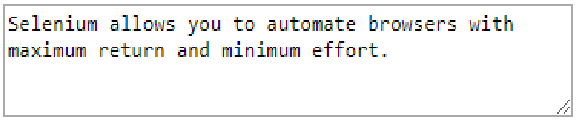
<textarea> … </textarea>

Defines a multiline input control. The text area can hold an unlimited number of characters. For example, the following lines will render as follows:



```
<textarea rows="4" cols="50">
Selenium allows you to automate browsers with
maximum return and minimum effort.
</textarea>
```

Figure 8: HTML Code for Multiline Input Control



Selenium allows you to automate browsers with
maximum return and minimum effort.

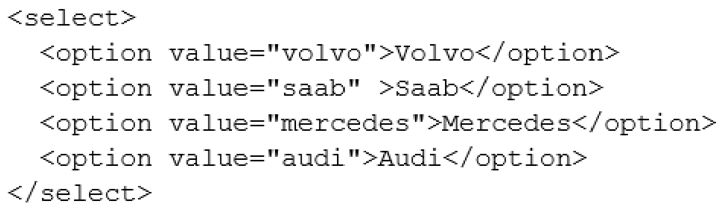
Figure 9: Multiline Input Control

<button>

Defines a clickable button.

<select> … </select>

Defines a drop-down list. When used with the **<option>…</option>** tags, the author can define a drop-down list and populate it as follows:



```
<select>
  <option value="volvo">Volvo</option>
  <option value="saab" >Saab</option>
  <option value="mercedes">Mercedes</option>
  <option value="audi">Audi</option>
</select>
```

Figure 10: Code for Select Control

The above code will render the following drop-down control:

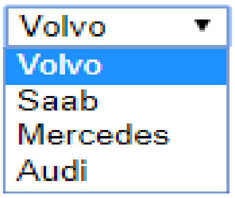
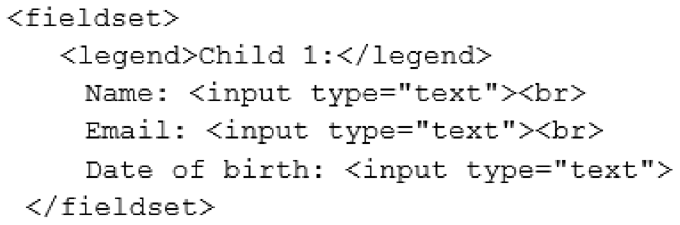


Figure 11: A Drop-down List

<fieldset> … </fieldset>

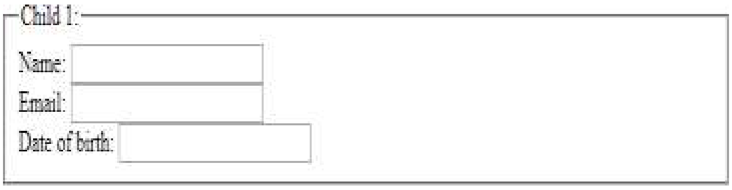
These tags allow the author to group related elements into a form. When used with the tag, **<legend>**, it draws a named box around controls which are deemed to be related as shown:



```
<fieldset>
   <legend>Child 1:</legend>
    Name: <input type="text"><br>
    Email: <input type="text"><br>
    Date of birth: <input type="text">
 </fieldset>
```

Figure 12: Fieldset Code

The above code will generate the following control set:



Child 1:

Name:

Email:

Date of birth:

Figure 13: Elements Grouped on Form

2.1.2 Understanding XML

XML (eXtensible Markup Language) is a markup language that is used to define rules for formatting documents in a way that is machine readable but is also highly readable by humans. XML was designed to accentuate simplicity and usability across the World Wide Web. While it is used in documents, XML allows the representation of arbitrary data structures which can be designed on the fly.

HTML was designed to display data with a specific focus on how the data looks. XML was designed to be a software and hardware-independent tool which can be used to transport and store data in a legible format.

XML tags are not predefined like HTML tags are. Instead, tags are specified by the creator of the XML document to any standards they want to use. The format of the tags is much like HTML. For example, here is a set of fields in XML:

```
<?xml version="1.0" ?>
<note>
      <date>2018-06-12</date>
      <hour>10:30</hour>
       <to>Francis</to>
       <from>Morrow</from>
       <body>Please pick me up this weekend!</body>
</note>
```

Figure 14: XML Sample Code

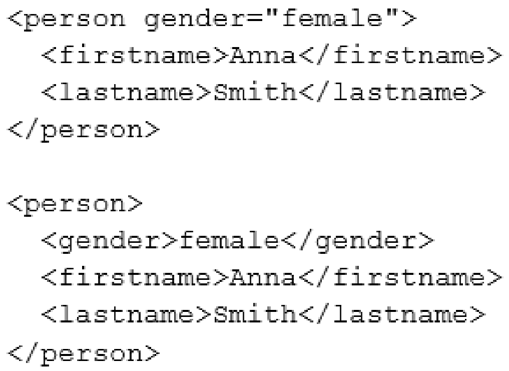
Note that each opening tag, **<from>**, has a matching closing tag, **</from>**. The total construct is called an element.

Sections can be embedded in other sections as shown. An XML document always forms a tree structure. The first element in the above figure shows that this is an XML document.

In addition to tags, XML supports attributes which supply extra information about the element they are associated with. An attribute consists of a pair of terms separated by an equal sign. For example:

<person *gender="female">*

The attribute is contained within the element's brackets. Rather than using an attribute, the same information can be used as an element. For example, the following two examples contain exactly the same information.



```
<person gender="female">
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>

<person>
  <gender>female</gender>
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

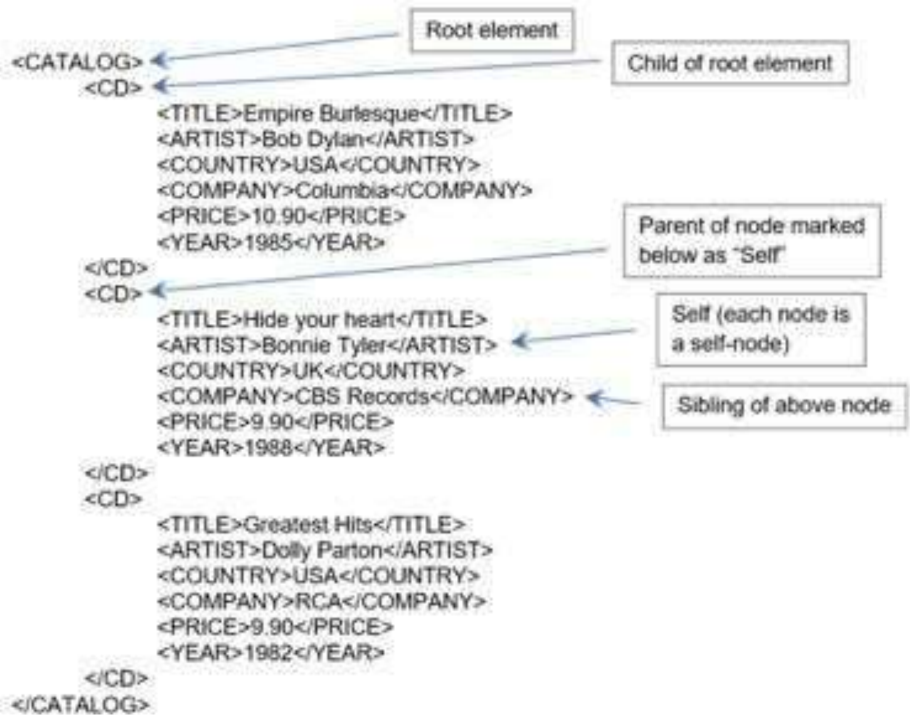
Figure 15: Attribute vs. Element

Attributes are not as flexible as elements. For example, the following points are stressed by W3C, the group that controls the XML standard:

- attributes cannot contain multiple values (elements can)
- attributes cannot contain tree structures (elements can)
- attributes are not easily expandable (for future changes)

Different computers store data in different ways; most of those ways are incompatible. XML allows these different computers to share data because the XML data is stored in plain text format. There is no need for complex handshakes between computers because they can communicate using text files.

XML separates data from the way it is presented. Therefore, the same XML data can be presented any way the author wants to.



```
<CATALOG>
    <CD>
        <TITLE>Empire Burlesque</TITLE>
        <ARTIST>Bob Dylan</ARTIST>
        <COUNTRY>USA</COUNTRY>
        <COMPANY>Columbia</COMPANY>
        <PRICE>10.90</PRICE>
        <YEAR>1985</YEAR>
    </CD>
    <CD>
        <TITLE>Hide your heart</TITLE>
        <ARTIST>Bonnie Tyler</ARTIST>
        <COUNTRY>UK</COUNTRY>
        <COMPANY>CBS Records</COMPANY>
        <PRICE>9.90</PRICE>
        <YEAR>1988</YEAR>
    </CD>
    <CD>
        <TITLE>Greatest Hits</TITLE>
        <ARTIST>Dolly Parton</ARTIST>
        <COUNTRY>USA</COUNTRY>
        <COMPANY>RCA</COMPANY>
        <PRICE>9.90</PRICE>
        <YEAR>1982</YEAR>
    </CD>
</CATALOG>
```

Root element
Child of root element
Parent of node marked below as "Self"
Self (each node is a self-node)
Sibling of above node

Figure 16: XML Database (Partial)

Since XML always describes a tree format, we can define specific relationships between elements. These relationships can be defined as follows:

- The **current** node is any arbitrary node that we choose. All other relationships derive from the current node. In the figure above, we have chosen “Bonnie Tyler” as the current node. All relationships then refer to that node.
- Each node can identify itself as **self.**
- A **parent** node is always one level higher in the hierarchy than the node selected as current. Each element node and attribute has exactly one parent (except for the root element.)
- A **child** node is always one level below its parent in the hierarchy.
- A **sibling** node is on the same level as the current node, under the same parent.
- An **ancestor** node is one in a direct path from the current node up through its parent, grandparent, great-grandparent, etc.
- A **descendent** node is one in a direct path down from the current node in the hierarchy (i.e., a child, a child of a child, etc.)

2.2 XPath and Searching HTML Documents

As mentioned above, to automate with Selenium, an automator must be able to locate any given object or element in an XML document. One of the most powerful ways to locate specific elements is to use XPath.

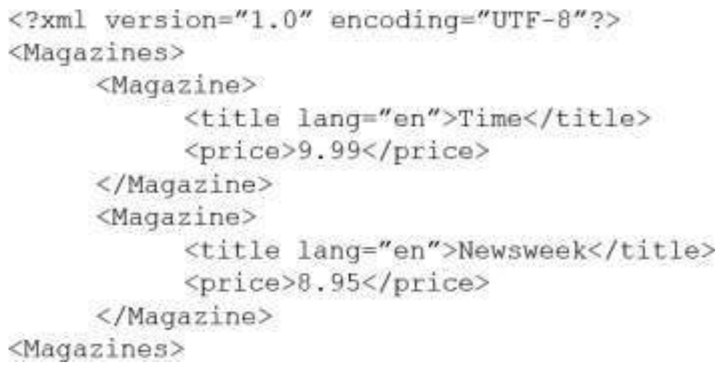
XPath uses path expressions to identify and navigate nodes in an XML document. Since HTML is a subset of XML, XPath can also be used to search HTML documents. These path expressions are related to the path expressions that might be used with a traditional computer file system which allow a user to walk through a folder hierarchy.

The following expressions will select nodes:

Table 6: XPath Expressions

Expression	Description
TheNode	Selects all elements with name “TheNode”
/	Selects from the root element
//	Returns descendants of the current element
.	Return the current element
..	Selects the parent of the current element
@	Selects an attribute of the current element

Below is a sample XML document and some examples of XPath usage.



```
<?xml version="1.0" encoding="UTF-8"?>
<Magazines>
      <Magazine>
            <title lang="en">Time</title>
            <price>9.99</price>
      </Magazine>
      <Magazine>
            <title lang="en">Newsweek</title>
            <price>8.95</price>
      </Magazine>
<Magazines>
```

Figure 17: Sample (partial) XML Database

Below are some path expressions and the results from them.

Table 7: Path Expressions

Path Expression	Result
Magazines	Selects all nodes with the name “Magazines”
/Magazines	Selects the root element “Magazines”
Magazines/Magazine	Selects all Magazine elements of Magazines
//Magazines	Selects all Magazine elements in the document
Magazines//Magazine	Selects all Magazine elements that are descendants of Magazines
//@lang	Selects all attributes named lang

Predicates are used to find a specific element or an element that contains a specific value. Predicates are always surrounded by square brackets and appear directly after the element name.

Table 8: Path Expressions Using Predicates

Path Expression	Result
/Magazines/Magazine[1]	Selects the first Magazine element
/Magazines/Magazine[last()]	Selects the last child Magazine element
/Magazines/Magazine[last()-1]	Selects the second to last Magazine element
//title[@lang]	Selects all title elements with attribute “lang”
//title[@lang=’en’]	Selects all title elements with attribute lang=en

XPath has wildcards which can be used to select XML nodes based on specified criteria.

Table 9: Wildcards in XPath

Wildcard	Description
*	Matches any element node
@*	Matches any attribute node
Node()	Matches any node of any kind

There is a rich diversity of operators that can be used in XPath expressions.

Table 10: Operators in XPath

Operator	Description	Example
|	Selects multiple paths	//Magazine | //CD
+	Addition	2 + 2
-	Subtraction	5 – 3
*	Multiplication	8 * 8
div	Division	14 div 2
mod	Modulus (remainder after division)	7 mod 3
=	Equal	price=4.35
!=	Not equal	price!=4.35
<	Less than	price<4.35
<=	Less than or equal to	price<=4.35
>	Greater than	price>4.35
>=	Greater than or equal to	price>=4.35
or	Or	price>3.00 or lang=”en”
and	And	price>3.00 and lang=”en”
not	Invert	not lang=”en”
||	String concatenation	“en” || “glish”

Likewise, there are many useful string manipulation functions available in XPath. Functions may be called with the namespace prefix “fn:”. However, since the default prefix of the namespace is “fn:”, string functions normally do not need the prefix.

Table 11: String Functions in XPath

Name	Description
string(arg)	Returns string value of the argument
substring(str, start, len)	Returns a substring of a string of length “len” starting at “start”
string-length(str)	Returns the length of the string. If there is no argument passed in, returns length of current node
compare(str1, str2)	Returns -1 if str1 < str2, 0 if strings are equal, +1 if str1 > Str2
concat(str1, str2, …)	Returns a concatenation of all strings inputted
upper-case(str)	Converts the string argument to upper case
lower-case(str)	Converts the string argument to lower case
contains(str1, str2)	Returns TRUE if str1 contains str2
starts-with(str1, str2)	Returns TRUE if str1 starts with str2
ends-with(str1, str2)	Returns TRUE if str1 ends with str2

A useful link for testing XPath expressions can be found at:

https://www.freeformatter.com/xpath-tester.html

2.3 CSS Locators

There are a lot of opinions as to whether CSS is a programming language or not. CSS stands for Cascading Style Sheets and is used mainly to prescribe how the various HTML elements in a set of HTML documents should render, on the screen, on paper, or in other media. External CSS style sheets are stored in CSS files.

HTML is a markup language and CSS is a style sheet language. While HTML and CSS give an author very powerful tools for displaying materials, most experts do not consider them as real programming languages.

However, when applied to Selenium testing, CSS is very useful in finding HTML elements, so that testing the browser can be automated.

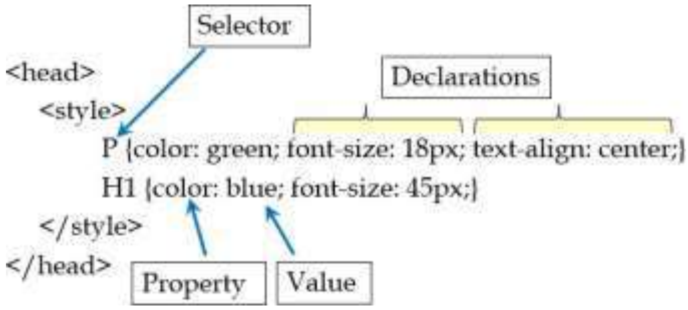
CSS can be used in HTML documents in three different ways:

1. An external style sheet: each HTML page must include a reference to the external style sheet file inside the **<link>**element which goes inside the **<head>**section
2. An internal style sheet: when a single HTML page is to have a unique style; the styles are defined within the **<style>**element, inside the **<head>**section of the document
3. An inline style: applies to a specific element, and is added directly to the element as an attribute

When multiple CSS styles are defined for the same element, the value from the last read style sheet will be used. Therefore, the order a style will be used will be defined (starting at the top) as follows:

1. Inline style (as an attribute inside an HTML element)
2. External and internal style sheets defined in the head section
3. The browser default value

A CSS rule-set consists of selector(s) and declaration block(s) as follows:



```
<head>
  <style>
    P {color: green; font-size: 18px; text-align: center;}
    H1 {color: blue; font-size: 45px;}
  </style>
</head>
```

Figure 18: CSS Rule-Set

Each selector points to an HTML element that you want to style. The declaration block contains one or more declarations separated by semicolons. Each declaration includes a CSS property name and a value separated by a colon. Declaration blocks are surrounded by curly braces.

CSS selectors are used to find elements in the HTML document based on element name, ID, class, attribute, or other specifiers. In many cases, XPath can be used in the same way to find certain elements, as shown in the table below:

Table 12: CSS vs. XPath Selectors

CSS	XPath	Result
div.even	//div[@class="even"]	Elements *div* with an attribute *class="even"*
#login	//*[@id="login"]	An element with *id="login"*
*	//*	All elements
input	//input	All input elements
p,div		All *p* and all *div* elements
div input	//div//input	All *input* elements inside all *div* elements
div > input	//div/input	All *input* elements that have the *div* element as the parent
br + p		Selects all *p* elements that are placed immediately after element *br*
p ~ br		Selects all *p* elements that are placed immediately before element *br*

As might be expected, CSS selectors also work with attributes.

Table 13: CSS selectors for attributes

CSS	Result
[lang]	All elements with the *lang* attribute
[lang=en]	All elements with the *lang* attribute of exactly *en*
[lang^=en]	All elements with the *lang* attribute starting with the string *en*
[lang | =en]	All elements that have the *lang* attribute equal to *en* or starting with the string *en* followed by a hyphen
[lang$=en]	All elements that have the *lang* attribute ending with the string *en*
[lang~=en]	All elements that have the lang attribute whose value is a whitespace-separated list of words, one of which is exactly the string *en*
[lang*=en]	All elements that have *lang* attribute containing string *en*

When dealing with form elements, there are a variety of CSS selectors available:

Table 14: CSS selectors for form elements

CSS	Result
:checked	Selects all checked elements (for check boxes, radio buttons, and options that are toggled to an *on* state
:default	Selects any form element that is the default among a group of related elements
:defined	Selects all elements that have been defined
:disabled	Selects all elements that have been disabled
:enabled	Selects all elements that have been enabled
:focus	Selects the element currently with focus
:invalid	Selects any form elements that fail to validate
:optional	Selects form elements which do not have *required* attribute set
:out-of-range	Selects any *input* elements whose current value is outside the *min* and *max* attributes
:read-only	Selects elements which are not editable by user
:read-write	Selects elements which are editable by user
:required	Selects form elements with required attribute set
:valid	Selects form elements that do validate successfully
:visited	Selects the links that a user has already visited

Finally, there are a variety of CSS selectors that may be useful to an automator, including:

Table 15: CSS Selectors Useful for Automation

CSS	Result
:not(<selector>)	Selects the elements that do not match the specified selector
:first-child	Selects all elements that are first children of their parent elements

:last-child	Selects all elements that are last children of their parent elements
:nth-child(<n>)	Selects all elements that are *nth* children of their parent elements
:nth-last-child(<n>)	Selects all elements that are *nth* children of their parent elements counting from the last child
div:nth-of-type(<n>)	Selects all element that are *nth div* children of their parent elements

Understanding HTML, CSS, XML, and XPath are essential to being successful with Selenium. What we have presented here is just the tip of the iceberg.

The W3C (World-Wide-Web Consortium) is an international community which works with many organizations and the public to define and develop web standards, including XML and XPath.

While not connected with W3C, there is an excellent set of no-cost tutorials on a web site called W3Schools. Tutorials include HTML, XML, XPath, CSS, and cover many other technologies that are useful when performing Selenium testing.

W3Schools and all of its tutorials are the copyright property of Refsnes Data. A4Q has no connection whatsoever with W3Schools of Refsnes Data. The following link will connect to the tutorials:

https://www.w3schools.com/default.asp

Chapter 3 - Using Selenium WebDriver

Keywords

Learning Objectives for Using Selenium WebDriver

STF-3.1 (K3) Use appropriate logging and reporting mechanisms
STF-3.2 (K3) Navigate to different URLs using WebDriver commands
STF-3.3 (K3) Change window context in web browsers using WebDriver commands
STF-3.4 (K3) Capture screenshots of web pages using WebDriver commands
STF-3.5 (K4) Locate GUI elements using various strategies
STF-3.6 (K3) Get state of GUI elements using WebDriver commands
STF-3.7 (K3) Interact with GUI elements using WebDriver commands
STF-3.8 (K3) Interact with user prompts in web browsers using WebDriver commands

3.1 Logging and Reporting Mechanisms

Automated test scripts are software programs that execute commands on the SUT. In doing this, they simulate humans running tests using the keyboard and mouse. Within the TAS there must be a mechanism that implements the Test Execution Layer. One way of implementing such a mechanism is writing test automation scripts from scratch. Such scripts can be run as any other Python script.

Rather than completely creating such scripts, we can leverage existing unit test libraries that can run tests and report their results, e.g., unittest or pytest. In this syllabus we have chosen pytest as our test execution library.

Pytest is a testing framework that makes it easier to write tests for Python, and more to our interest, for Selenium WebDriver using Python. Pytest makes it easy to write small tests but scales up to support writing complex automation suites.

When invoked, Pytest executes all tests in the current directory or its subdirectories. It specifically looks for all files which matches the patterns: “test__*.py” or :*__test.py” and then executes them.

When run with no flags (**pytest**), pytest simply runs all tests that are found in directories below it. Alternately, flags may be set to modify its behavior as follows:

- (**pytest -v**) Verbose mode which displays full test names rather than just a period
- (**pytest -q**) Quiet mode which displays less output
- (***pytest --html=report.html***) Run test(s) with a report to the HTML file

Tests can be marked to be treated in a special way. For example, when you put (**@pytest.mark.skip)** before a test definition, pytest will not run the test. If you put (**@pytest.mark.xfail***)* before a test definition, it informs the run-time engine that the test is expected to fail. Such tests, if they fail when run, will not be reported in the same way as tests which fail when they are expected to pass.

When a manual tester runs a scripted test and it fails, the tester has a pretty good idea of what happened. They know exactly what steps led up to the failure, exactly what happened during the failure, what data was used, and what the failure looked like. When running an exploratory test, while the tester does not have a script for the exact steps taken, they have a general theory of execution and the limited ability to backtrack and see why a failure occurred.

In automation, almost none of that is true.

Often, in automation, the error message that is recorded by the tool is wholly insufficient for the person who is reviewing a failed execution to understand what happened. Frequently the error message has no context; the recorded error may have nothing to do with the actual failure which stopped the execution of the test.

For example, suppose a failure occurs at step ***N***. Depending on the way the failure occurred, the result might mean the interface of the SUT is not left in the correct state. As the script tries to execute step ***N+1***, the step fails because the SUT is not in the correct state to perform the step. The log reports that it was step ***N+1*** that failed.

Troubleshooting the problem then starts off incorrectly. Since the test analyst who ran the automated test (usually in batch mode) may not have direct knowledge of the actual test that was run, trying to troubleshoot it to determine if it is an actual SUT failure or an automation failure can take a lot of time.

It does not have to be this way in automation. Good logging can help a test analyst determine quickly if the failure was caused by the SUT or not. It is essential that an automator pays attention to the logging of the test. Good logging may make the difference between a failing automation project, and one which adds a lot of value to the organization.

Logging is a way to track events that occur during execution. The automator can add logging calls to a script to record information to facilitate understanding the execution. Logging may be done before a step is taken and after a step is taken and may include the data that was used in the step and the behavior that occurred as a result of the step. The more logging that is done, the better you can understand the result of the test.

In some cases when you are testing safety-critical or mission-critical software, detailed logging may be required for audit purposes.

Logging may be conditional. That is, data may be saved to an intermediate location and only placed in the formal log if a failure occurs or if a complete troubleshooting record is needed. Logging of every step may not be desirable when the test runs as expected; however, in case of failure, that logging information may save hours of troubleshooting by recording, step by step, exactly what happened during execution and what data was used.

Not every automation project needs such robust logs. Often, an automation project will start out with one or two automators who will create and run their own scripts. In such a case, the logging described above may be seen as overkill. However, what must be noted is that when small automation projects succeed, it is pretty much guaranteed that management will want more: a lot more automation. The more success, the higher the demand for more.

That means that eventually, the project will reach such size that the logging described ***will*** be required. At that point, existing tests would need to be retrofitted with better logging. It is more effective and efficient to start logging robustly from the start.

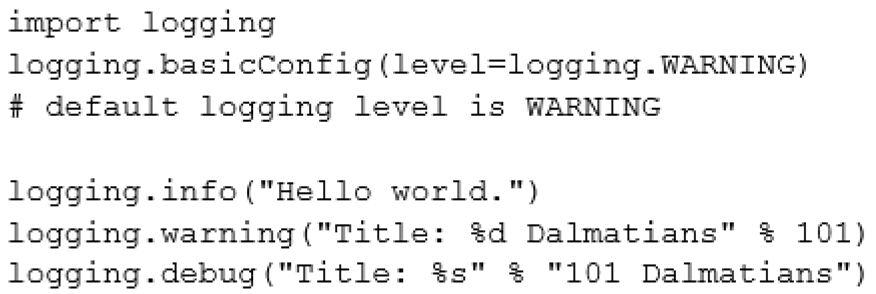
Python has a very robust set of logging facilities that may be used with WebDriver. At any point in an automated script, the automator may add logging calls to report out any information desired. This can include general information for later tracing the test (e.g., “I am about to click on the XYZ button”), warnings about something that happened that does not rise to the level of a failure (e.g., “Opening file <ABC> took longer than expected.”) or actual errors, raising an exception which triggers end of test events such as cleaning up the environment and moving on to the next test.

The Python logging library has five different levels of messages that may be saved. From lowest to highest levels:

- DEBUG: for diagnosing problems
- INFO: for confirmation that things worked
- WARNING: something unexpected occurred, a potential problem

- ERROR: a serious problem occurred
- CRITICAL: a critical problem occurred

When a log is printed out to the console or a file, the message level can be set to one of those five settings (or custom ones can be created) allowing the user to see just the messages desired. For example, if the console is set to level WARNING, the user will not see DEBUG or INFO messages, but will see WARNING, ERROR, and CRITICAL logging messages. For example, assume the following code were run:



```
import logging
logging.basicConfig(level=logging.WARNING)
# default logging level is WARNING

logging.info("Hello world.")
logging.warning("Title: %d Dalmatians" % 101)
logging.debug("Title: %s" % "101 Dalmatians")
```

Figure 19: Logging Example

Because the default logging level is WARNING, the following would be printed to the console:

```
WARNING:root: Title: 101 Dalmatians
```

When running a test case, it is important that something actually be tested. Every test case needs to have expected results and behaviors that we can check. Python has a built in device to check whether the correct data or behavior occurred: the assertion.

An assertion is a statement that is expected to be true at a particular point in the script. For example, if testing a calculator, we could add two plus two and then assert that the answer should equal four. If for some reason the calculation is incorrect, the script will throw an exception.

The syntax is as follows:

assert sumVariable==4, “sumVariable should equal 4.”

With Selenium WebDriver, a step in a test case often consists of the following actions.

1. Locate a web element on a screen.
2. Act on the web element
3. Ensure the correct thing happened.

For action one (1), if the element is not found, or is found in the wrong state, an exception is thrown.

For action two (2), if the attempt to act on the web element fails, an exception is thrown.

For action three (3), we can use an assertion to check that the expected behavior occurred, or value was received.

This, in essence, follows the way a manual tester would perform this step and allows the person evaluating a failed test to understand what occurred.

Reporting is often associated with logging, but it is different. Logging supplies information about the automation execution to the test analyst who ran the suite, and to the automator(s) who are responsible for repairing the tests when needed. Reporting is supplying that information, and likely other contextual information to the various stakeholders, outward and upward, who want or need it.

It is important that the automators determine who wants or needs reports and what information they are interested in. Simply sending the unedited logs to the stakeholders would likely overwhelm many of them by burying them in information they neither need nor want.

One way to deal with reports is to create them from the logs and other information and make them available to the stakeholders such they can download them when they want. The other way is to create and send the reports as soon as they are ready to the stakeholders who want them.

Either way, it is a good idea to automate report creation and dissemination if at all possible to remove yet another manual task that needs to be done.

Reports should contain a summary with an overview of the system being tested, the environment(s) the tests were run in, and the results that occurred during the run. As mentioned, each stakeholder may want a different view of the results and those needs should be supplied by the automation team. Often, stakeholders may want to see trends of the testing, not just a point in time. Since each project likely has different stakeholders with different needs, the more the reporting tasks are automated, the easier it will be.

3.2 Navigate to Different URLs

3.2.1 Starting a test automation session

There are many different browsers that you might want to test. While the basic task of a browser is to allow you to view and interact with various web pages, each browser is likely to behave a bit differently than other browsers.

In the early days of the internet, there were some pages that only worked in Netscape, others that only worked correctly in IE. Luckily, most of those problems are long gone, although there still might be some incompatibilities between browsers. When an organization wants to deliver a web site, testing must still be done on different browsers to ensure compatibility.

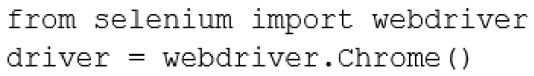
In chapter 1.4, when we first introduced Selenium WebDriver, we mentioned that different browsers required different drivers to ensure that the automated scripts we write work with the different browsers. Following is the list we first showed there:

- Chrome (chromedriver.exe)
- Internet Explorer (IEDriverServer.exe)
- Edge (MicrosoftWebDriver.msi)
- Firefox (geckodriver.exe)
- Safari (safaridriver)
- HtmlUnit (HtmlUnit driver)

For example, if we wanted to test the Chrome browser, we would first need to download the Chrome driver (conveniently named ***chromedriver.exe***) and install that file in a well-known place on the test workstation. This is often done by editing the ***Path*** environmental variable so that your operating system will find it when needed.

Please note that, like so much of technology, these driver files and the information needed to use them, change rapidly. The information in this syllabus has been verified and is valid right now at the time it is being written. All bets are off for tomorrow. The best thing an automator can do is to become acquainted with the help documentation on the various support web sites for the various browsers, and the Selenium WebDriver support site.

To work with web pages, you first need to open a web browser. This can be achieved by creating a ***WebDriver object***. Instantiating the ***WebDriver object*** will create the programming interface between a script and the web browser. It will also run a WebDriver process if one is needed for particular web browser. Usually it will run the web browser itself, as well.

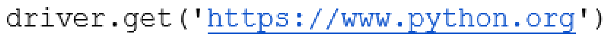


```
from selenium import webdriver
driver = webdriver.Chrome()
```

Figure 20: Code to Create WebDriver object

This code will only work after the ***chromedriver.exe*** is installed as described above.

Creating a WebDriver object will start a web browser with an empty page. To navigate to desired site page, the function ***get()*** should be used. Note that the driver object’s functionality is accessed by using dot notation as Python is an object-oriented language.

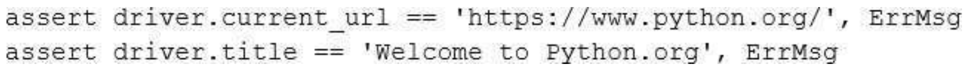


```
driver.get('https://www.python.org')
```

Figure 21: Driving to URL

It is also possible to attach a WebDriver object to an existing webdriver process or create a WebDriver process and make it attach to an already running browser through Selenium RemoteWebDriver, but these operations are beyond the scope of this syllabus.

After opening a page or navigating to a different one it is advisable to check if the correct page has been opened. The WebDriver object contains two useful attributes to do that: ***current_url*** and ***title***. Checking values of these fields enables script to keep track of its current page.

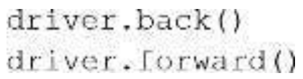


```
assert driver.current_url == 'https://www.python.org/', ErrMsg
assert driver.title == 'Welcome to Python.org', ErrMsg
```

Figure 22: Assert Correct Page

3.2.2 Navigating and refreshing pages

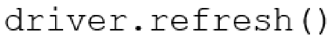
When you need to simulate navigating forward and backward in the web browser, you should use ***back()*** and ***forward()*** methods of the WebDriver object. They will send appropriate commands to the WebDriver. These methods have no arguments.



```
driver.back()
driver.forward()
```

Figure 23: Navigating Browser History

It is also possible to make the web browser refresh the current page. This can be done by calling the ***refresh()*** method of WebDriver.



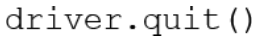
```
driver.refresh()
```

Figure 24: Refreshing the Browser

3.2.3 Closing the browser

At the end of the test you need to close web browser process and any additional driver processes that have been run. If you fail to close the browser it will stay open even when testing has finished. It is a good idea to try to set the environment to the same state that your test(s) started in. That allows an indefinite number of tests to be run unattended.

To close the browser controlled by webdriver call the ***quit()*** method of the webdriver object.

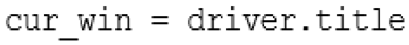


```
driver.quit()
```

Figure 25: Closing the Entire Browser

You need to place the ***quit()*** function in the part of test script that is run regardless of the result of the test. A common mistake is to treat closing the browser as one of the ordinary test steps. In such a case when the test fails, a test runner library stops the execution of the test steps and the step which is responsible for closing the browser is never run. Usually test libraries have their proprietary mechanisms of defining and running teardown code (e.g., the ***tearDown()*** methods of Python ***unittest*** module). Please refer to the documentation of the library you are using to get more details on this subject.

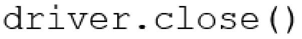
If you have multiple tabs open in the browser under test, you can determine which of the tabs is open by checking the title of the current window using:



```
cur_win = driver.title
```

Figure 26: Getting Active Window Title

If you do not want to quit the whole browser and only close tab in the browser, use the above function and move through the tabs until you get to the window to close. Then close the tab using the ***close()*** method. When the last open tab is closed, the browser process automatically quits.



```
driver.close()
```

Figure 27: Closing the Active Tab

This method takes no parameters and closes active tab. Remember to switch context to the desired tab to be able to close that tab. Once the close happens, calling any other WebDriver command other than ***driver.switch_to.window()*** will raise a ***NoSuchWindowException*** since the object reference still points to the window that no longer exists. You must proactively switch to another window before calling any WebDriver methods. After you switch tabs, you can then check the title to determine which tab is now active. We will discuss controlling multiple tabs below.

3.3 Change the Window Context

Sometimes, when testing more complex applications or scenarios, you will need to change the current context of the GUI you are testing. This may be due to the need of checking the test outcome in a different system or to execute a test step in different application.

Sometimes the GUI of the application you are testing will be so complex that you will need to switch between frames or windows.

A web browser does not need to have focus to be able to execute Selenium Webdriver commands, as WebDriver protocol is based on HTTP communication. This enables WebDriver to run several tests simultaneously or to control several browsers in one script.

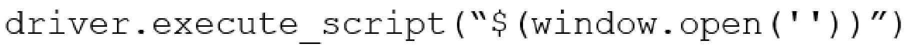
Changing the script context can be done in three ways:

- changing browsers
- changing windows/tabs within one browser
- changing frames within one page

To open two browsers, you need to create two WebDriver objects. Each WebDriver object controls one browser. Each WebDriver object is then used in the test script in accordance with the steps of the automated test case. WebDriver objects can control the same type of web browsers (e.g., both

Chrome, both Firefox) or different types (e.g., one Chrome and the other Firefox). You may also open more than two browsers if you need. Remember that each browser is controlled by one webdriver object. And remember to close all browsers at the end of the test.

Opening up multiple tabs in a single browser can get complicated, since different browsers and operating systems have different methods for doing that. One way that works for Windows in the Chrome browser is to execute a function call in JavaScript code as follows:



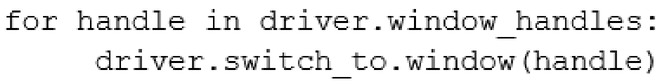
```
driver.execute_script(“$(window.open(''))”)
```

Figure 28: JavaScript Call to Open New Tab

Further discussion about opening multiple tabs in one web browser is outside of the scope of this syllabus.

To switch between open tabs in a browser you first need to get the list of all open tabs (windows). This list can be found in the ***window_handles*** attribute of the WebDriver object. Be aware that the order of handles in the ***window_handles*** array may be different that the order of tabs in the browser.

You can cycle through all of the window tabs by using the following code:

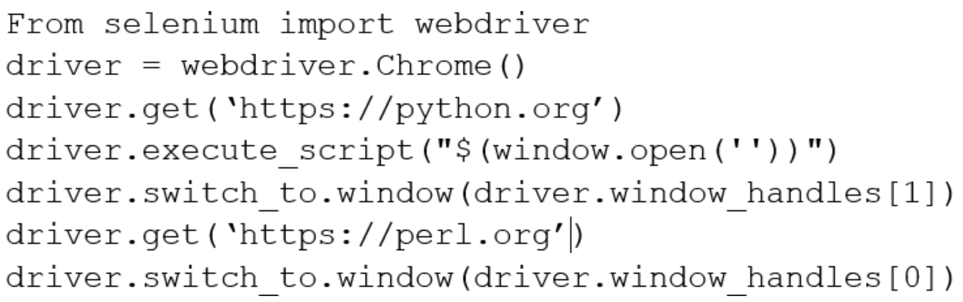


```
for handle in driver.window_handles:
     driver.switch_to.window(handle)
```

Figure 29: Cycling Through the Open Windows

The safest way to determine which window is the currently open window is to use the ***driver.title*** attribute mentioned earlier.

The following Python code opens Python’s homepage, opens up a new tab, then opens Perl’s homepage in the second tab, and then switches the web browser back to the tab containing Python’s homepage:



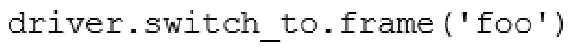
```
From selenium import webdriver
driver = webdriver.Chrome()
driver.get(‘https://python.org’)
driver.execute_script("$(window.open(''))")
driver.switch_to.window(driver.window_handles[1])
driver.get(‘https://perl.org’)
driver.switch_to.window(driver.window_handles[0])
```

Figure 30: Open Two Tabs and Switch Between Them

Please note this code is not safe for an actual production test as it assumes the tab's handles will be in order. It is only for reference purposes. To actually see this code run, add some sleep() functions so it does not switch too fast.

The third situation when you may want to change context in a web browser is changing frames. This is frequently required; if you do not change context to a particular frame you will not be able to find elements in that frame, and consequently you will be not able to automate tests for the elements in that frame.

To change context to a specific frame, you first need to find that frame. If you know the ***ID*** of the frame, you can switch straight to it, as follows:



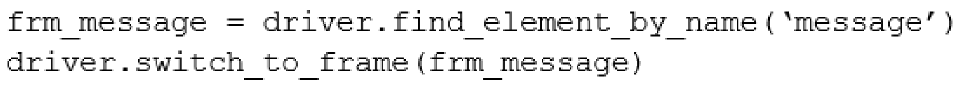
```
driver.switch_to.frame('foo')
```

Figure 31: Switching Frame

where ***foo*** is the ***ID*** of the frame you want to change context to.

If the frame has no ID or you want to use different strategy to find it, switching context can be done in two steps. First step, as mentioned earlier, is finding the frame as a web element, and the second step is switching to the found frame. Switching in this case is executed by the same method as switching by id, but the argument given to a function is the found web element.

Following is a section of Python code which shows an example of frame switching where finding the frame is the first step:

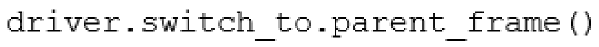


```
frm_message = driver.find_element_by_name('message')
driver.switch_to_frame(frm_message)
```

Figure 32: Finding then Switching Frame

Note that during the call to ***switch_to.frame()***, we do not use apostrophes nor quotation marks, as we pass the ***frm_message*** variable as a variable argument rather than a string containing the ID of the frame.

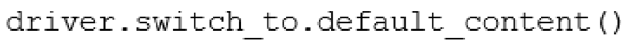
If you want to switch back to the parent frame, use:



```
driver.switch_to.parent_frame()
```

Figure 33: Switching to Parent of a Frame

You can also switch back to the whole page by using:

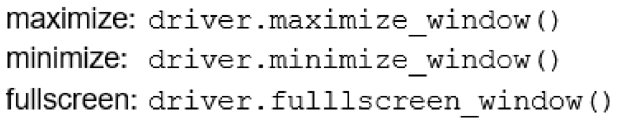


```
driver.switch_to.default_content()
```

Figure 34: Switching to Main Window

Apart from changing a window context or a frame context of the web browser, the Selenium WebDriver framework allows you to manipulate the size of a web browser's window. The term window here is used as the operating systems notion of the entire window, rather than just a tab inside a web browser.

Selenium allows minimizing and maximizing the web browser and putting it into full screen mode. Python bindings for this functionality are shown below:



```
maximize: driver.maximize_window()
minimize: driver.minimize_window()
fullscreen: driver.fulllscreen_window()
```

Figure 35: Sizing the Browser

These functions do not take any arguments, because they operate on the web browser controlled by the ***driver*** object.

3.4 Capture Screenshots of Web Pages

When a manual tester is running a test case, they interact with the GUI objects on the screen visually. If a test case fails, they can tell because what they see on the screen is no longer correct.

A tester using automation does not have that luxury.

Test automation scripts are not able to reliably check the layout and appearance of web pages. There are many times when it is useful to take screenshots of the screen or a particular element of the screen and to save them off to the log or to a known location, so they can be viewed at a later time.

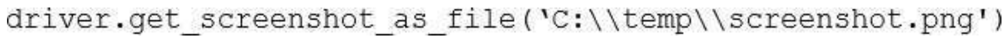
- When the automated script detects that a failure occurred
- When the test is highly visual, and the pass/fail determination can only be made by viewing the screen image
- When dealing with safety- or mission-critical software which might require an audit of the testing
- When doing configuration testing on different systems

To take advantage of the information that screenshots provide, they have to be taken at the right moment. Often, the parts of test scripts that take screenshots are placed just after test steps that control the user interface or in tear down functions. However, since they can be an invaluable tool for understanding the automated testing, they can be taken anywhere.

One important issue to deal with is naming the files with unique names and locations so that your automation does not overwrite screenshots taken earlier in the run. There are a variety of different ways to do this; however, they are beyond the scope of this syllabus to elaborate.

Screenshots can be taken in two different scopes: the entire browser page, or a single element in the browser page[1]. Both of these use the same method call but are called from different contexts.

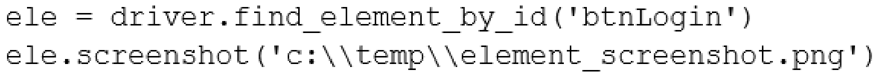
The method call to use is ***screenshot()***. The following Python example shows how to take a screenshot of a whole page and place it in a specific location:



```
driver.get_screenshot_as_file(‘C:\\temp\\screenshot.png')
```

Figure 36: Saving Screenshot of Entire Page

This example shows how to take screenshot of an element and saving it to a specific location:



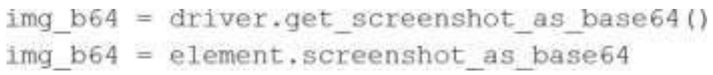
```
ele = driver.find_element_by_id('btnLogin')
ele.screenshot('c:\\temp\\element_screenshot.png')
```

Figure 37: Saving Screenshot of Element

Taking a screenshot in the browser can take some time as a lot of processing must be done by the workstation. If the GUI state is rapidly changing (e.g., by concurrently executing AJAX libraries), the screenshot taken may not show the exact state of the page or the element that was expected. Again, there are remedies for this situation, but they are beyond the scope of this syllabus.

If you want to take a screenshot and treat it as something other than a file, there are alternatives in WebDriver. For example, suppose rather than using an HTML file for a log, you want to log to a database instead. When you take a snapshot of the screen, or of an element, you would not want to create a ***.png** file with it, you might want to stream it directly to a database record. The following calls would create an image as a base64 encoded string version of the snapshot. The base64 encoded version is much safer to stream than a binary file, such as a ***.png** file. The first call will capture the entire window, the second a single element:

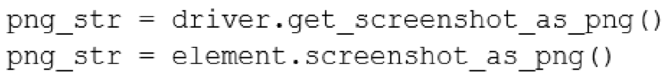
[1] Documentation of the WebDriver Python library version 3.13.0 claims that the functionality for taking a screenshot of a WebElement is available by calling the function: WebElement.screenshot(‘Filename.png’). The authors of this syllabus have not been able to verify that this functionality works in the Chrome 67 browser with chromedriver 2.36, although it is defined in the WebDriver W3C technical recommendation. If you need this functionality, there are workarounds documented on several websites. Try a Google search for “Python WebDriver screenshot of WebElement”. This function does appear to work when testing the Firefox browser.



```
img_b64 = driver.get_screenshot_as_base64()
img_b64 = element.screenshot_as_base64
```

Figure 38: Creating Base64 String from Image

Likewise, if you want to get a binary string representing an image, without saving it to a file, you can use the following calls to return the binary representation of a ***.png*** file. The first call will return the image of an entire screen, the second will return an image of a single element.



```
png_str = driver.get_screenshot_as_png()
png_str = element.screenshot_as_png()
```

Figure 39: Creating Binary String from Image

3.5 Locate GUI Elements

3.5.1 Introduction

To perform most operations with WebDriver, you will need to locate the UI elements on which you want to operate in the currently active screen. This can be accomplished by using the ***find_element_*** or ***find_elements_*** methods within WebDriver. Both of these methods have different versions depending on what you want to search by. For example, all of the following are available for searching:

- ***by_id (id_)***
- ***by_class_name (name)***
- ***by_tag_name (name)***
- ***by_xpath (xpath)***
- ***by_css_selector (css_selector)***

In each case the argument taken by the function is a string representing the element(s) we are trying to find. The return values are different; for example, the singular version (***find_element***_) will return a single WebElement (if one is found) while the multiple version (***find_elements_***)will return a list of WebElements that match the argument.

To have this discussion, we need to introduce a concept used in web development and testing: the DOM (Document Object Model). When a web page is loaded into the browser, the browser creates a DOM, modeling the web page as a tree of objects. This DOM defines a standard for accessing the web page. As defined by the W3C:

> *"The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document."*

The DOM defines:

- All HTML elements as objects
- The properties of all HTML elements
- The methods that can be used to access all HTML elements
- The events that affect all HTML elements

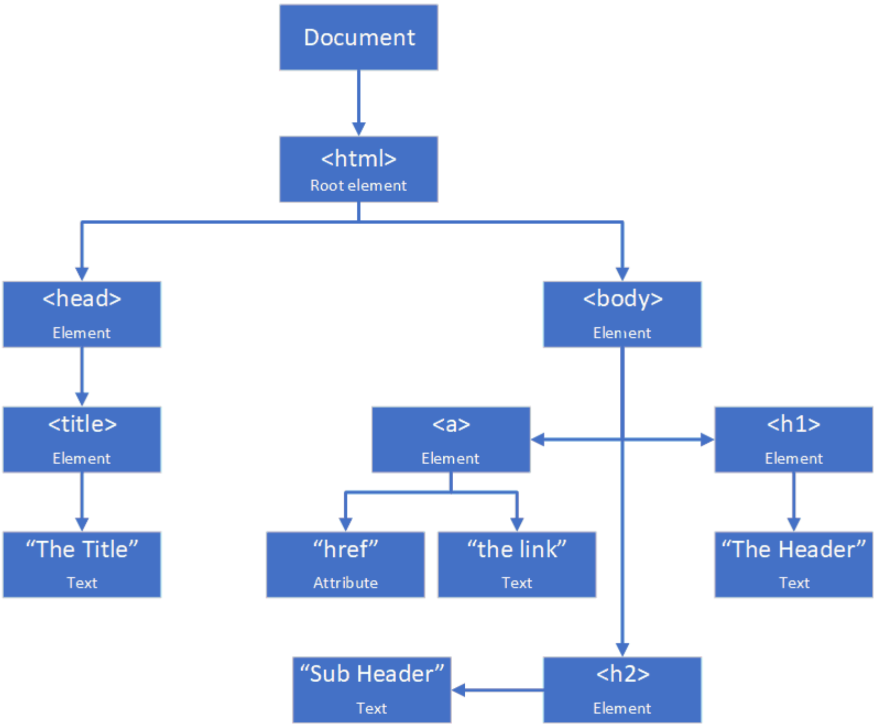
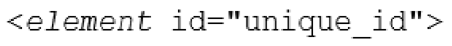


Figure 40: Document Object Model Tree

3.5.2 HTML Methods

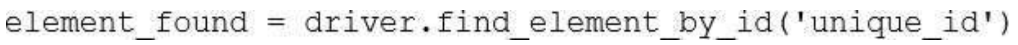
The following methods depend on finding screen elements by searching the document for HTML artifacts. The first method we will discuss is using the HTML ***ID*** attribute. for each different way of locating a WebElement, there are liable to be both advantages and disadvantages.



```
<element id="unique_id">
```

Figure 41: HTML Snippet

Python WebDriver locating example:



```
element_found = driver.find_element_by_id('unique_id')
```

Figure 42: Locate by ID

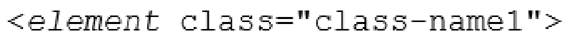
Advantages:

- Efficient
- By definition, each ID should be unique in the HTML document
- A tester can add IDs to the SUT easily (note: these changes should be peer reviewed)

Disadvantages:

- IDs can be auto-generated, meaning that they can change dynamically
- IDs are not appropriate for code which is used in multiple places, e.g., a template used to generate headers and footers for different modal dialogs
- A tester may not be allowed to modify the SUT

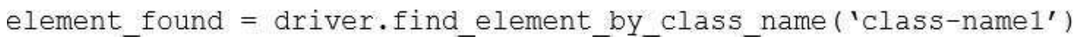
The second way to find a WebElement is to find it by searching for its class name. This refers to the HTML *class* attribute on a DOM element, for example, in this HTML snippet:



```
<element class="class-name1">
```

Figure 43: HTML Snippet

Python WebDriver locating example:



```
element_found = driver.find_element_by_class_name(‘class-name1’)
```

Figure 44: Locate by Class

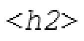
Advantages:

- Class names can be used multiple places in the DOM, but you can limit the locating to the loaded page (for example on a popup modal dialog)
- A tester can add class names to the SUT easily (note: these changes should be peer reviewed)

Disadvantages:

- Since class names can be used in multiple places, more care must be taken not to locate the wrong element
- A tester may not be allowed to modify the SUT

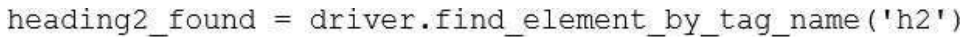
The third way to discuss finding a WebElement is to use the HTML tag name of the element. This refers to the DOM Tag Name of an element, for example, in this HTML snippet:



```
<h2>
```

Figure 45: HTML Tag

Python WebDriver locating example:



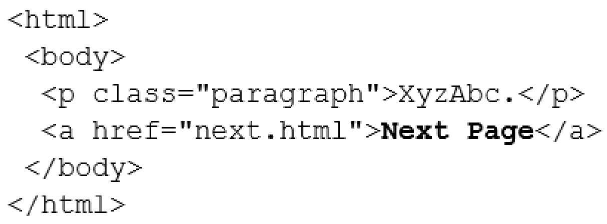
```
heading2_found = driver.find_element_by_tag_name('h2')
```

Figure 46: Locate by Tag

Advantage: If a tag is unique to a page, you can narrow where you look.

Disadvantage: If a tag is *not* unique to a page, you may locate the wrong element.

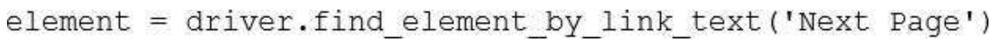
Our fourth way to identify the WebElement by the link text. This refers only to an anchor tag which contains text that will be highlighted for a user to click on. There are actually two similar methods: you can specify the entire text string or a part of the string. For example, in the following HTML snippet, the link text has been bolded to identify it.



```
<html>
 <body>
  <p class="paragraph">XyzAbc.</p>
  <a href="next.html">Next Page</a>
 </body>
</html>
```

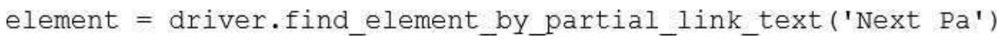
Figure 47: HTML Snippet

Python WebDriver locating examples:



```
element = driver.find_element_by_link_text('Next Page')
```

Figure 48: Locate by Link Text



```
element = driver.find_element_by_partial_link_text('Next Pa')
```

Figure 49: Locate by Partial Link Text

Advantages:

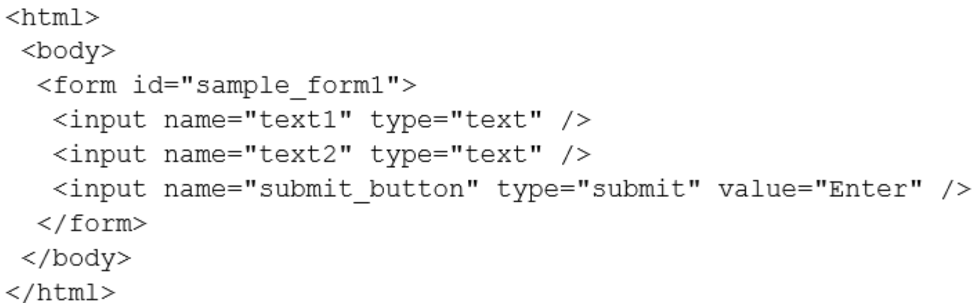
- If the link text is unique to a page, you can find the element
- The link text is visible to the user (in most cases), so it's easy to know what the test code is looking for
- Partial link text is slightly less likely than the full link text to change

Disadvantages:

- The link text is more likely to change than an ID or class name
- Using partial text may make it harder to uniquely identify a single link

3.5.3 XPath Methods

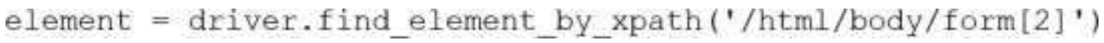
As discussed in section 2.2, XPath (XML Path Language) is a language that can be used to select specific nodes using a variety of criteria in an XML document. For example, consider the following HTML fragment. Since HTML is a subset of XML, it can be searched using XPath. There may be issues if the HTML is poorly formed (e.g., not all close tags are included).



```
<html>
 <body>
  <form id="sample_form1">
   <input name="text1" type="text" />
   <input name="text2" type="text" />
   <input name="submit_button" type="submit" value="Enter" />
  </form>
 </body>
</html>
```

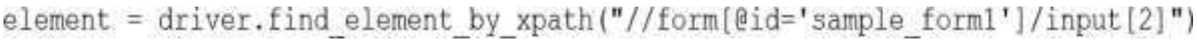
Figure 50: HTML Snippet

WebDriver can use XPath to find a specific node, and from that, an element can be located. You could specify an absolute path, but that is a bad idea because any change would likely invalidate your code. A better way is to identify a relative path starting from a found node that matches a criterion. Below is an example of specifying an absolute path, and then a more robust version finding the desired element via XPath. Both will return the second input field in the HTML snippet.



```
element = driver.find_element_by_xpath('/html/body/form[2]')
```

Figure 51: Using XPath with Absolute Path

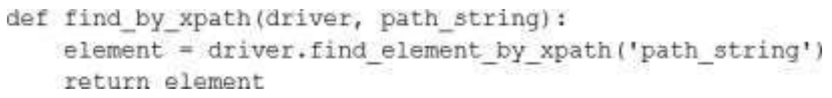


```
element = driver.find_element_by_xpath("//form[@id='sample_form1']/input[2]")
```

Figure 52: Using XPath with Relative Path

Using the relative path will often take more keystrokes to define, but it will fail to find the desired node far less often. That is a pretty good trade off.

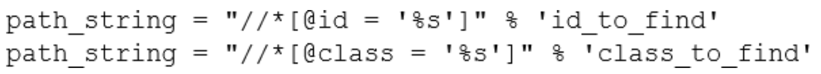
Within an XPath string, you can look for ***id***, ***name***, ***class***, ***tag name***, etc. Thus, it's possible to create generic locator functions using XPath by passing in an attribute type to the function, or a path string that incorporates the attribute in it. For example:



```
def find_by_xpath(driver, path_string):
    element = driver.find_element_by_xpath('path_string')
    return element
```

Figure 53: : Generic XPath Find Function

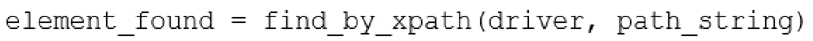
The variable ***path_string*** can then be assembled based on the attribute we are trying to find. Below are two examples, the first searching by ***id***, the second searching by ***class***:



```
path_string = "//*[@id = '%s']" % 'id_to_find'
path_string = "//*[@class = '%s']" % 'class_to_find'
```

Figure 54: Building an XPath String

Calling the generic function would then be performed as follows:



```
element_found = find_by_xpath(driver, path_string)
```

Figure 55: Calling Generic XPath Function

This is inelegant, but it demonstrates the flexibility of XPath for locating elements.

Advantages:

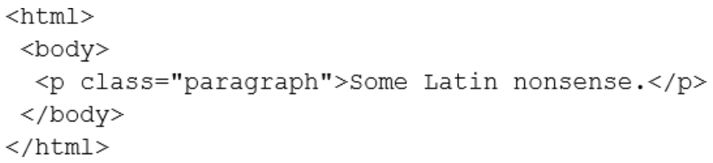
- You can find elements that don't have any unique attributes (id, class, name, etc.)
- You can use XPath in generic locators, using the different "By" strategies (by id, by class, etc.) as needed

Disadvantages:

- Absolute XPath code is "brittle" and can break with the smallest change to the HTML structure
- Relative XPath code can find the wrong node if the attribute or element that you look for is not unique on the page
- Because XPath may be implemented differently between browsers, extra effort may be required to run WebDriver tests in those environments

3.5.4 CSS Selector Methods

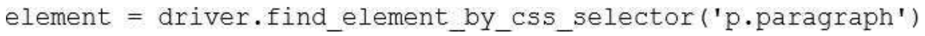
As we discussed in section 2.3, we can also find elements using CSS selectors. For example, in this HTML snippet:



```
<html>
 <body>
  <p class="paragraph">Some Latin nonsense.</p>
 </body>
</html>
```

Figure 56: HTML Snippet

By using the rules of CSS Selectors, found in section 2.3, the following code should identify the expected node:



```
element = driver.find_element_by_css_selector('p.paragraph')
```

Figure 57: Identifying WebElement via CSS Selector

Advantage: If an element is unique to a page, you can narrow where you look.

Disadvantage: If an element is *not* unique to a page, you may locate the wrong element.

3.5.5 Locating Via Expected Conditions

Selenium with Python bindings has an ***expected_conditions*** module that can be imported from ***selenium.webdriver.support*** with several convenient predefined conditions. You can create custom ***expected_condition*** classes, but the predefined classes should satisfy most of your needs. These classes offer greater specificity than the locators mentioned above. That is, they do not simply determine if an element exists, they all check for a specific state for that element to be in. For example, ***element_to_be_selected()*** not only determines that the element exists, but it also checks to see if it is in a selected state.

While this list is not complete, some examples are:

- alert_is_present
- element_selection_state_to_be(element, is_selected)
- element_to_be_clickable(locator)
- element_to_be_selected(element)
- frame_to_be_available_and_switch_to_it(locator)
- invisibility_of_element_located(locator)
- presence_of_element_located(locator)
- text_to_be_present_in_element(locator, text_)
- title_is(title)
- visibility_of_element_located(locator)

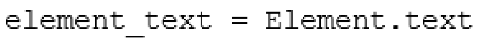
We will discuss these further in section 4.2, since many of these are also used as wait mechanisms.

3.6 Get the state of GUI elements

Simply getting the location of a specific WebElement is often not enough. For test automation, there may be several different reasons why we also need to access some information about an element. Information might include its current visibility, whether it is enabled or not, or whether it is selected or not. Reasons may include:

- Ensure the state is as expected at a given point in the test
- Make sure a control is in a state that it can be manipulated as needed in the test case (i.e., enabled)
- Make sure that after the control was manipulated, it is now in the expected state
- Make sure that the expected results are correct after a test is run

Different WebElements have different ways of accessing information from them. For example, many WebElements have a text property which can be retrieved by using the following code:

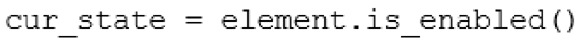


```
element_text = Element.text
```

Figure 58: Retrieving Text Property

Not every WebElement has a text property. For example, if we examined a header node (perhaps by using a ***find_by_css_selector***(‘h1’) method), we would expect it to have a text property. Other WebElements may not have the same property. The context of WebElement and how it is used can help you understand what properties it is likely to have.

Some WebElement properties must be accessed using a WebElement method. For example, suppose you wanted to determine whether a particular WebElement is currently enabled or disabled. You could call the following method to get that Boolean value:



```
cur_state = element.is_enabled()
```

Figure 59: Check if WebElement is Enabled

The following table is not comprehensive, but it lists many of the common properties and accessor methods that you might need for automation.

Table 16: Common Properties and Accessor Methods

Property/Method	Arguments	Returns	Description
get_attribute()	property to retrieve	property, attribute or None	Gets property. If no property by that name, gets attribute of that name. If neither, returns None
get_property()	property to retrieve	property	Gets property.
is_displayed()		Boolean	Returns true if visible to user

is_enabled()	Boolean	returns true if element is enabled
is_selected()	Boolean	Returns true if checkbox or radio is selected
location	X,Y location	Returns X,Y location on the renderable canvas
size	Height, Width	Returns the Height and Width of the element
tag_name	The tag_name property	Returns the tag_name of the element
text	Text for the element	Returns the text associated with the element

3.7 Interact with UI elements using WebDriver commands

3.7.1 Introduction

So, you have located the WebElement that you wanted your automated test to interact with (discussed in section 3.5). You have made sure that the element is in the correct state, so you can manipulate it as per your test case (discussed in section 3.6). Now it is time to actually make the change you wanted.

One of the main reasons that graphical user interfaces became popular is that there are a limited set of controls which can be manipulated; each control essentially is an on-screen metaphor that can be easily manipulated using the keyboard and/or mouse. Each control is designed to be easily understood, even by novice computer users. So, I can type into a text field, click on a radio button, select an item from a list box, etc.

Automating the manipulation of these controls, however, can be more difficult to understand. What a tester does manually is often done unconsciously. As automators, we need to make sure we understand all the nuances of making changes to on-screen controls.

In the upcoming sections, we will discuss manipulating the following:

- Text fields
- Clickable WebElement (fields you can click on)
- Check boxes
- Drop-down lists

For any WebElement you plan on manipulating, you may care about several things:

- The WebElement exists
- The WebElement is displayed

- The WebElement is enabled

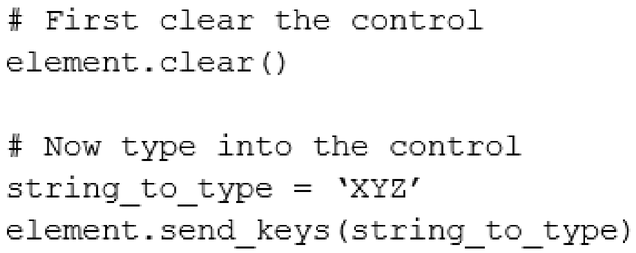
Depending on the web site, the way the HTML was written, whether Ajax is being used, and any number of other conditions, there may be a discrepancy as to whether the WebElement that you want to deal with actually needs to be displayed or not for you to manipulate it. For example, if the WebElement is on the active page and is enabled, but has been scrolled out of view, trying to manipulate it may not work. Tests run in Chrome have shown that, on some pages, working with the WebElement not currently visible works and on other sites it throws an exception. Our best advice is to try to get all WebElements that you work with visible (displayed) on the screen.

Since a browser screen may be dynamically modified (via Ajax, for example) or updated based on previous actions, these checks likely should be done using the ***expected_condition*** classes which allow us to synchronize timing; we will discuss those in chapter 4.

3.7.2 Manipulating Text Fields

When typing into an editable text field, you will generally want to first clear the element, then type the string desired into the control. We will assume that you have already checked that the element exists, is displayed, and is enabled. If you don't ensure the state of the control, and one or more of them is untrue, your attempted manipulation of the element will cause an exception.

We will assume that the you have already located the input control, and it is in the variable named *element*.



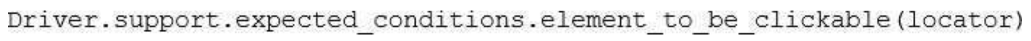
```
# First clear the control
element.clear()

# Now type into the control
string_to_type = 'XYZ'
element.send_keys(string_to_type)
```

Figure 60: Type into an Edit Field

3.7.3 Clicking on WebElements

Clicking on an element simulates a mouse click. This can be done on a radio button, link, or image; essentially any place that you might click with your mouse manually. We are not including checkboxes here; those will be discussed in the next section. Once again, it might be important to check to make sure that the WebElement is currently clickable. you can use the method, ***element_to_be_clickable,*** from the ***expected_condition*** class to wait for it to become clickable.

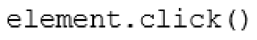
Again, we assume that the WebElement has been located. We may need to wait to ensure that it is ready to be clicked, using an ***expected_condition*** method:



```
Driver.support.expected_conditions.element_to_be_clickable(locator)
```

Figure 61: Synchronization Method

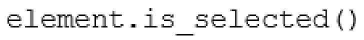
Synchronization will be discussed in chapter 4. Assuming the WebElement has been referenced by the variable element and it is clickable, we then call:



```
element.click()
```

Figure 62: Click on WebElement

If the WebElement were a link or a button, we would expect that a context change would occur that could be verified in the screen. If this was a checkbox, the result may be selected or unselected: this will be discussed in the next section. If this was a radio button being clicked, however, you could verify that the button is actually selected by calling:



```
element.is_selected()
```

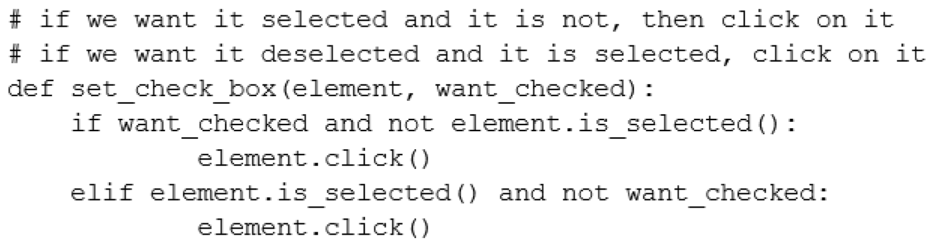
Figure 63: Checking State of WebElement

3.7.4 Manipulating Checkboxes

While we click on check boxes, they must be treated differently than other clickable controls. If you click on a radio button multiple times, it remains *selected*. However, every time you click on a checkbox, it toggles the *selected* state. Click once, now *selected*. Click again, now not *selected*. Click again and it goes back to *selected*.

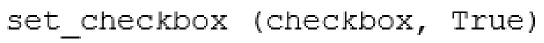
Therefore, it is important to understand the state that we want to achieve when manipulating a checkbox. We can write a function which will take the checkbox and the desired state and perform the right action no matter the current state the checkbox is in.

Assume the checkbox has been loaded into the variable *checkbox*. Assume a Boolean variable *want_checked* passed in to determine the final state we want.



```
# if we want it selected and it is not, then click on it
# if we want it deselected and it is selected, click on it
def set_check_box(element, want_checked):
    if want_checked and not element.is_selected():
            element.click()
    elif element.is_selected() and not want_checked:
            element.click()
```

Figure 64: Function to Set Checkbox



```
set_checkbox (checkbox, True)
```

Figure 65: Calling the Checkbox Function to Check Box

3.7.5 Manipulating Dropdown Controls

Dropdown list boxes (select controls) are used by many websites to allow users to select one of many options. Some of the dropdown boxes allow multiple items from the list to be selected concurrently.

There are many ways to work with the list of a select control. These include options for selecting single items or multiple items. There are also different ways to deselect items.

Selection options include:

- Search through the HTML to find the item desired and then click on it
- Select by an item value (***select_by_value***(*value*))
- Select all items that display matching text (***select_by_visible_text***(*text*))
- Select an item by index (***select_by_index***(*index*))

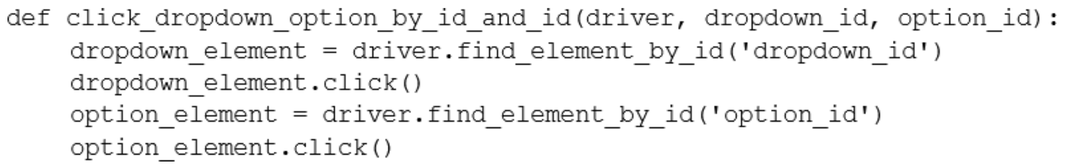
Deselection options include:

- Deselect all items (***deselect_all***())
- Deselect by index (***deselect_by_index***(*index*))
- Deselect by value (***deselect_by_value***(*value*))
- Deselect by visible text (***deselect_by_visible_text***(*text*))

Once you are done selecting/deselecting the items in the drop down list, there are several options to allow you to see what has been selected:

- Return a list of all selected items (***all_selected_options***)
- Return the first selected (or only if a single select control) (***first_selected_option***)
- See a list of all the options in the list (***options***)

Since clicking on list items is a common task we need to do, we can build a function like we did with checkboxes. In this case, we need to first click on the dropdown button to reveal the full list. Then, once the list opens, find the desired option and click on it. Here is a function that would do that.



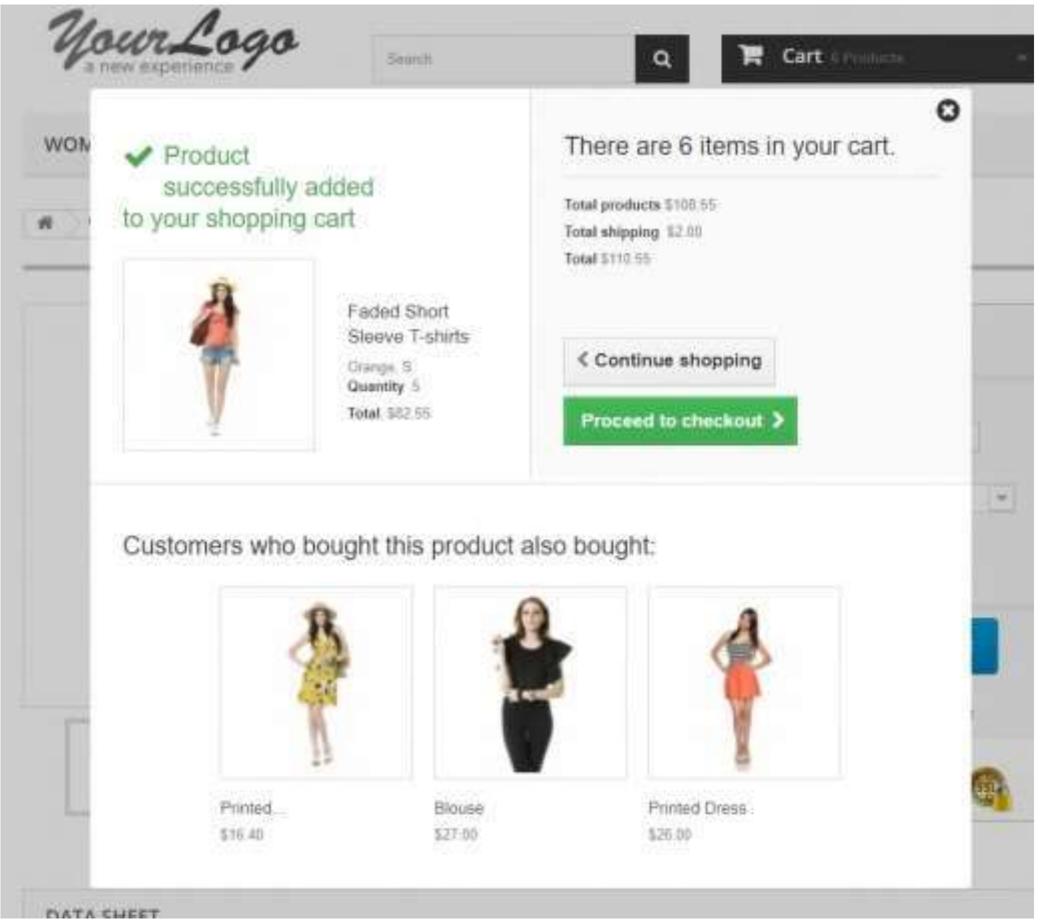
```
def click_dropdown_option_by_id_and_id(driver, dropdown_id, option_id):
    dropdown_element = driver.find_element_by_id('dropdown_id')
    dropdown_element.click()
    option_element = driver.find_element_by_id('option_id')
    option_element.click()
```

Figure 66: Function to Click on Dropdown List Item

3.7.6 Working with Modal Dialogs

A modal dialog is a box that pops up over a browser window and does not allow access to the underlying window until it has been handled. These are similar to user prompts dialogs, but different enough to discuss separately. We will discuss user prompts, such as alerts, in the next section.

Generally modal dialogs are invoked when the author of the web site wants to get specific input from the user (e.g., a user name/password pop up) or have specific tasks performed by the user. For example, on an ecommerce site, adding an item to a cart may make an informational window pop up as shown below:



Product successfully added to your shopping cart

Faded Short Sleeve T-shirts

Orange, S

Quantity 5

Total $82.55

There are 6 items in your cart.

Total products $108.55

Total shipping $2.00

Total $110.55

Customers who bought this product also bought:

Printed... $16.40

Blouse $27.00

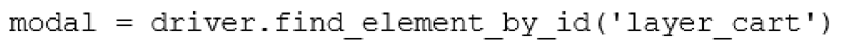
Printed Dress $26.00

Figure 67: Ecommerce Cart Modal Window

In the above example, the user may decide which way to go: proceed to the checkout or to continue shopping. In addition, any amount of extra information may be presented in the modal dialog. In this case, the ecommerce site is trying to entice the user to buy something else based on what they placed into the cart.

All of the code to the modal dialog is found in the HTML code that popped up the modal dialog. Therefore, manipulating the modal dialog is as simple as finding the code in the calling page and manipulating it. This follows the same rules for locating and manipulating the form controls as we discussed earlier.

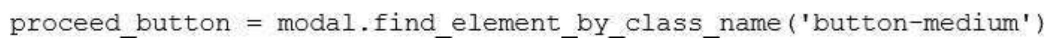
For example, suppose that we want to click on the *Proceed to Checkout* button in this modal dialog. The first step would be to determine the location of the code for the modal dialog. In this case, the ***ID*** of the section representing the modal element is ***layer_cart***. we would create a WebDriver object reference for this element of the code as follows:



```
modal = driver.find_element_by_id('layer_cart')
```

Figure 68: Finding the Modal Element

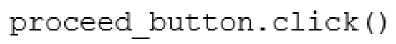
The next task would be to identify the element representing the button by using the *Inspect* functionality of the browser. In this case, the class name of the button is ***button_medium***. Once again, we want to create a reference to this element, so we can manipulate it. We can use the following code:



```
proceed_button = modal.find_element_by_class_name('button-medium')
```

Figure 69: Finding a Button in the Modal Element

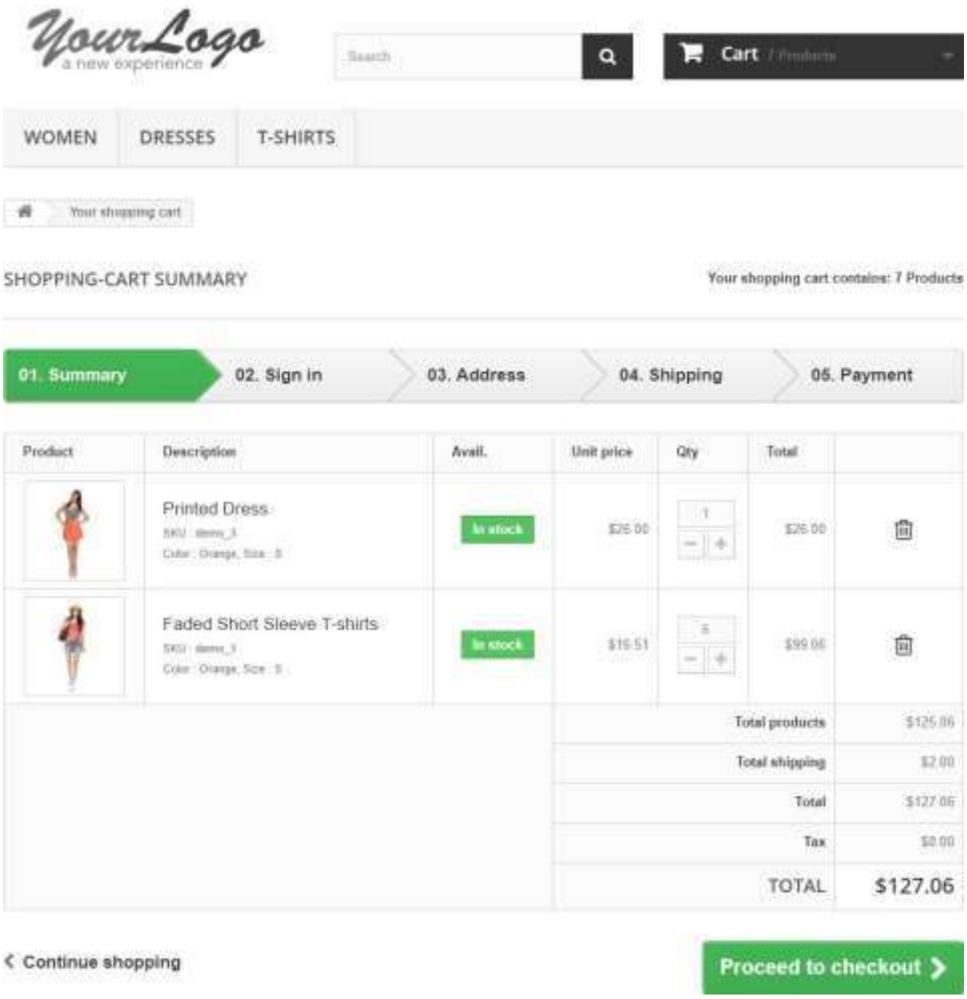
Once found, pressing the button is as easy as calling the click() method for that reference:



```
proceed_button.click()
```

Figure 70: Pressing Button

At this point, the modal dialog is closed, and we move to a new location in the main browser window, in this case: the shopping cart summary page as shown below:



SHOPPING-CART SUMMARY

Your shopping cart contains: 7 Products

Product	Description	Avail.	Unit price	Qty	Total	
	Printed Dress SKU : demo_3 Color : Orange, Size : S	In stock	$26.00	1	$26.00	
	Faded Short Sleeve T-shirts SKU : demo_1 Color : Orange, Size : S	In stock	$16.51	6	$99.06	
					Total products	$125.06
					Total shipping	$2.00
					Total	$127.06
					Tax	$0.00
					TOTAL	$127.06

Figure 71: Shopping Cart Screen

3.8 Interact with user prompts in web browsers using WebDriver commands

User prompts are modal windows requiring users to interact with them before the user can continue to interact with the controls in the browser window itself.

For automation purposes, user prompts are usually not handled automatically. Therefore, if your script tries to ignore the prompt and continue sending commands to the browser window itself, an ***unexpected alert open*** error will be returned by the action.

Each user prompt has an associated user prompt message (which may be NULL). This is a text field which may be picked up using the code shown below.

W3C defines three different alert-type dialogs:

- Alert

- Confirm
- Prompt

Since they are all defined such that they work almost the same, we will discuss the alert prompt.

The alert dialog box is often used to make sure that the user is made aware of some meaningful information.

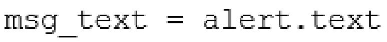
Since alert dialog boxes are not actually a part of the web page, they require special handling. WebDriver using Python has a set of methods which allow you to control the alert dialog from your automation script. These methods are common to all three user prompt dialogs.

This first creates a reference using a method of the WebDriver object called ***switch_to***. The syntax to use this method follows:

```
alert = driver.switch_to.alert
```

Figure 72: Create an Alert Object

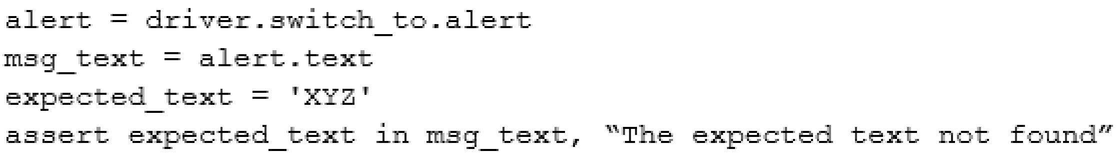
We can get the text in the alert via:



```
msg_text = alert.text
```

Figure 73: Getting Text from Alert

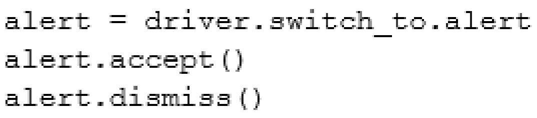
To determine if expected text is in the alert , switch to the alert, get the text, and then check to see if the expected text was in the alert. If the text is there, the variable ***passed*** will be set to ***True***. If not, the ***passed*** variable will be set to ***False***.



```
alert = driver.switch_to.alert
msg_text = alert.text
expected_text = 'XYZ'
assert expected_text in msg_text, "The expected text not found"
```

Figure 74: Comparing Alert Text

Alert dialogs may be closed in two ways. Consider the ***accept()*** method as pressing the OK button, and the ***dismiss()*** method as pressing the Cancel button, or the window close button in the upper corner of the prompt.



```
alert = driver.switch_to.alert
alert.accept()
alert.dismiss()
```

Figure 75: Closing an Alert

Chapter 4 - Preparing Maintainable Test Scripts

Keywords

fixture, Page Object Pattern, persona

Learning Objectives for Preparing Maintainable Test Scripts

STF-4.1	(K2) Understand which factors support and affect maintainability of test scripts
STF-4.2	(K3) Use appropriate wait mechanisms
STF-4.3	(K4) Analyze GUI of SUT and use Page Objects to make its abstractions
STF-4.4	(K4) Analyze test scripts and apply Keyword Driven Testing principles to building test scripts

4.1 Maintainability of Test Scripts

In chapter 1, section 2 we discussed the difference between a manual test and an automated script. We need to revisit that discussion because it has a direct influence on the maintainability of the software that we build called automation.

Reduced to its essential core, a manual test is a directed set of abstract instructions which are only valuable when used by a manual tester to actually run the test. The data and expected results should also be there, but the heart is in the steps to take. The manual tester adds context and reasonableness to those abstract statements, allowing tests of almost any complexity to be run successfully.

If a step in the manual script says to click on a button, the tester can do it with little thought. They do not stop to wonder if the control is visible, if it is enabled, or if it is the right control for the task. The fact is, however, that they do think of those things subconsciously. If the control is not visible, they can try to do something to make it visible. If not enabled, they won’t try to use the control blindly; they will try to figure out why it is disabled and see if they can fix it. Once they can get to the control, they can do the right thing. And, if something happens incorrectly with that control, they can identify what occurred, so they can write the defect report and/or modify the manual test procedure.

Every automation tool is dumb. No matter how expensive, each has very limited context and almost no reasonableness built into it. However, automators are human and really smart. We can add some context and reasonableness to a tool by programming them into our automated scripts.

However, the more we try to program that intelligence into the automated script, the more complex the script gets, and the more likely there will be failures of the script simply due to its innate complexity.

Writing automation for testing has often been compared to trying to shoot down a bullet with another bullet.

That saying has always meant there is a lot of complexity to automation. Complexity, however, is a dual-edged sword. We need to build intelligence into our scripts so that they better simulate a human tester running a test. We need to handle an ever-increasing amount of complexity in the SUTs that we work on.

But the more complexity we add to our automation, the more failures of the automation we can expect.

No matter how smart we are as automators, there are limits to what conditions we can control through the automated script.

A manual tester can try to figure out why that control is not visible. An automator probably can't; we are limited to putting code into the script to tell the script to wait for a finite time, hoping that the problem will resolve itself. The same with the control being enabled.

What an automator can do is try to make sure that the control is in the correct state before using it, and if it's not, wait a set amount of time, and if it is still not usable, log the error and close the execution of the script out gracefully so we can move onto the next test. If we can access the control, we can check to see if the control behaves as expected after we manipulate it. And, if there was a problem, again we can write a useful log message, so we can troubleshoot the issue more effectively and efficiently.

We would like to make our job easier. That means putting intelligence into callable functions rather than having to program intelligence into each script individually. In other words, move intelligence up to the TAA and/or TAF and out of the script.

By moving more intelligence upward and out of the scripts themselves, the scripts become more maintainable and more scalable. If our code which adds intelligence fails, it will make a lot of scripts fail, but fixing them all can be done at a single point of contact.

We gave you some examples of building these callable functions. The code we described, however, was not nearly good enough for production automation. Good automators tend to build these aggregate functions that some automators call wrapper functions. An example of a wrapper function with the intelligence built in follows in abstract form. Assume I want to click on a checkbox. I am going to try to build the function such that it mimics what the manual tester actually thinks and does.

The arguments to this wrapper function are liable to include: the checkbox to use, the final state desired (checked or unchecked), perhaps the amount of time we are willing to wait for it if it is not ready right away. Then the tasks would follow this logic:

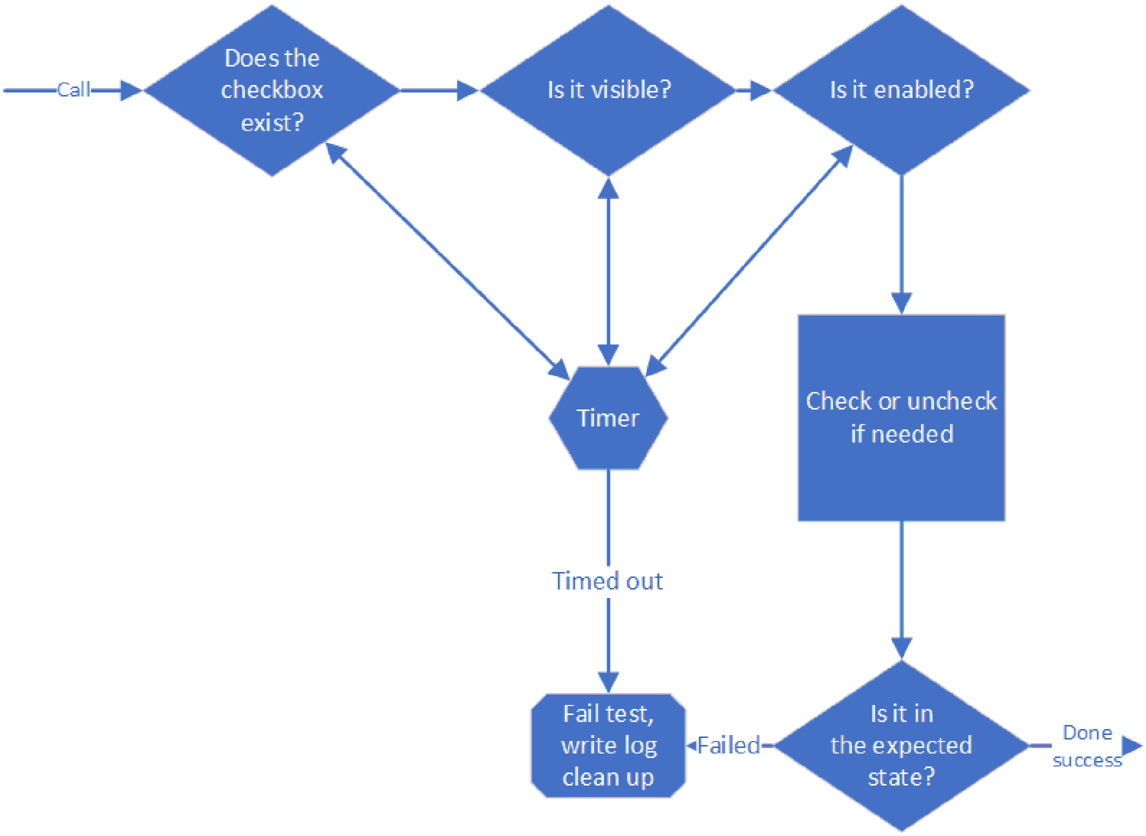


Figure 76: Wrapper Function Logic

A lot of effort should be put into the *failed* block. The log statements should be comprehensive such to cut down the troubleshooting effort when a test fails. The cleanup functionality should ensure that the test suite can continue on to the next, and the next after that, and the next after that test. A wise automator once said that the most important test that an automator needs to consider is the next one. If you can always run the next test, you can run the whole suite, no matter how many tests fail throughout the execution.

At the end of the day, testers are trying to find out where the SUT does not work correctly (and to build confidence when it does seem to work correctly). If our automation works well, we should expect failures to occur; we need to gather in the failure information and move on to the next test.

Remember that every test is built to examine the SUT, its environment, and its usage to tell us what we don't already know. A test that does not tell the tester something that they didn't know is not a valuable test whether it is automated or not. In the ISTQB® Certified Tester Foundation Level certification syllabus, they discuss test design techniques like equivalence class partitioning and boundary value analysis. Those techniques are designed to reduce the amount of testing to find out the essential information about the SUT as fast as possible.

Wrappers are important, but other functions can be built to enhance the automation. Any code that the automation needs to call repeatedly should be functionalized. This follows the good development practices of functional decomposition and refactoring. Create libraries of functions which can be included in your code to give yourself and your teammates tool boxes. Remember to move everything out of the actual scripts to these callable libraries wherever possible.

It takes time and resources to build these functions. If you want to be a successful automator, it might be the best investment you ever make.

When testing a browser, make sure when locating WebElements with XPath and CSS Selectors, you do not use absolute paths. As discussed in the previous chapter, any change to the HTML is likely to cause those paths to change, breaking the automation. While relative paths can also break, they tend to break less often and therefore need less maintenance.

Ensure that you discuss the automation with the developers in your organization. Let them know your pain. They can do many things in the way they write their code to prevent constant breakage of your automation that deals directly with the HTML.

Define global names (variables, constants, function names, etc.) that are meaningful. This makes the code more readable, which has a side effect of making maintenance on the automation code easier. Easier code maintenance usually means fewer regression defects when changes are made. It takes a few more seconds to come up with good names but can save hours on the back end.

Comments. Lots of comments. Meaningful comments. Many automators figure it is their code and they will remember why they wrote it the way they did. Or they figure that automation is ‘different’, not like real code. Those people are wrong. You can forget from one day to another the clever thing you did to solve a problem. Don’t force yourself to reinvent the wheel. When your automation project starts to succeed, other people will start putting their fingers into your code. Make it easy on them.

Create test accounts and fixtures that are specifically for the automation. Do not share these with manual testers. Automation needs to have certainty when it comes to data used in the automation; manual testers making changes are likely to break your automation.

These accounts and fixtures should not be tied into any specific individual. Generic accounts that mimic real accounts are much better to isolate them from changes. You should have enough different accounts that multiple tests should not interfere with each other’s data.

Create enough data in these accounts to make them simulate real accounts. Don't forget different personas for these accounts. If your SUT has different types of users (e.g., novices, techno geeks, power users, etc.) these should be modeled in your test accounts.

We discussed logging in chapter three, section one. Take logging seriously. If your automation is a success, your management will want more. A lot more. One of the ways to make automation more scalable is to do good logging from the start. You will not want to retrofit all of your hundred (thousands?) of scripts with good logging.

Python has a robust set of logging facilities; or, you can create your own. Think outside the box when considering logging. Solid logs can reduce time troubleshooting when failures occur. Consider the needs of your organization. If your SUT is mission- or safety-critical, your logs might need to be complete enough to be audited.

Try to model the thinking processes of the manual tester. What do they learn and take away from the SUT when they run a manual test? See if you can capture and log that same information. Remember, you can be as creative as you want determining the how, when and where of your logging.

Control the files you create. Use consistent naming conventions and save the files to consistent folders. Consider making a folder with a timestamp in the name and placing all files from a run in there. If multiple machines are running the automation, or if testing in different environments, consider adding the workstation name or environment names to the folder names.

Think of including time stamps in the file names. That will guarantee that your automation does not overwrite earlier test run results. If the files are not needed in the future, put them into directories that can be destroyed without harm. If the files need to be saved, make sure that your folder structure includes that information.

Use consistent file names. Meet with the other automators and come up with a style and naming convention agreement. Teach those standards and guidelines to new automators brought onto the team. It does not take long to build chaos when everyone does their own thing when creating persistent artifacts.

Automators traditionally have been the rebels of development. Code cowboys (and girls) who like to do their own thing and push the envelope. However, the investment in automation tends to be really large as a factor in both resources and time. Minimum standards and guidelines should be adopted to make sure that investment has at least a chance to pay off.

4.2 Wait Mechanisms

When a manual tester runs a test, the consideration of waiting is not really an issue. For example, suppose the tester opens a file. If the file opens in a timely manner (timely being defined by the tester), no further thought goes into it. Any time a manual tester clicks on a control or otherwise manipulates the SUT, there is an implicit clock in their head. If they think that something takes too long, they are likely to run the test again, paying explicit attention to the timing.

Something that has never happened, though, is a tester sitting for hours waiting patiently for a particular action to occur.

We have discussed several times in this syllabus about the context that a manual tester adds to a test. Timing is one of those contextual things we have to understand.

Suppose the tester is opening up a very small file: a couple hundred bytes. If it does not open instantaneously, the tester is likely to be concerned and may open up an incident report on it. Suppose that same tester tries to open a two-gigabyte file; if it takes thirty seconds to open they might not be surprised at all. If they get an informational message that the file is going to be slow to open, they would cancel out the message and continue waiting without batting an eye.

The context matters.

In automation, no matter the tool, it has little to no context built in.

Generally, the tool has a built in time it is willing to wait for an action to occur. If the expected action does not occur within that time frame, then the tool records a failure and it moves on (where it to moves depends on the tool, the setup, and a variety of other things).

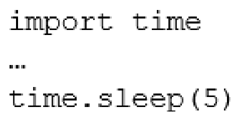
If the tool is set to wait 3 seconds, then an action that occurs in 3.0001 seconds—perhaps well within the contextual time range of the manual tester—will be considered a failure.

If the automator removes all timing consideration, then it is possible that the tool will literally wait forever for an action to occur. There are few worse feelings than coming in Monday morning and finding out that the automation suite you set to start Friday evening before leaving is on test two and has been sitting there since three minutes after you left on Friday.

An automator must understand the needs of their automation and the way their tool(s) work.

Selenium WebDriver with Python has several different wait mechanisms that an automator can use when setting up synchronization for their automation. One explicit wait mechanism should be used rarely yet is one of the most common ways many automators use.

The following code will likely be recognized by almost anyone who has ever automated any tests:



```
import time
…
time.sleep(5)
```

Figure 77: An Explicit Wait

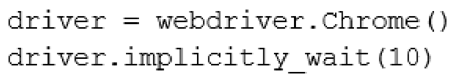
While there may be times that this is a good idea, most times it is not. We have seen times where automators have used so many of these that the automation runs slower than a manual tester running exactly the same test.

This type of wait mechanism is always aimed at the worst possible case. Since the worst possible case does not really occur that often, most of this time waiting is simply wasted.

For this course, we will assume that the sleep() function will only be considered for use when debugging an exercise; other than that, don't use it.

Selenium with WebDriver has two main types of waiting mechanisms: implicit waits and explicit waits. We will mainly focus on explicit waits, so we will dispose of implicit waits first.

An implicit wait in WebDriver is set when the WebDriver object is first created. The following code will create the driver and set the implicit wait:



```
driver = webdriver.Chrome()
driver.implicitly_wait(10)
```

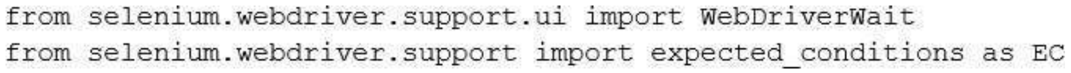
Figure 78: Setting Implicit Wait

The implicit wait, as defined above, will be in effect until the WebDriver is destroyed. This implicit wait instructs WebDriver to poll the DOM for a certain amount of time when trying to find an element which is not immediately found. The default setting is 0 seconds waiting. Any time an element is not found immediately, the above code tells WebDriver to poll for ten seconds, asking (conceptually) every so many milliseconds, "Are you there yet?" If the element shows up within that time period, the script continues to run. The implicit wait will work for any element or elements that are defined in the script.

Explicit waits require the automator to define (usually for a specific element) exactly how long WebDriver should wait for that particular element, often for a particular state of that element. As mentioned above, the extreme case of an explicit wait is the ***sleep()*** method. Using this method is much like using a chain saw rather than a scalpel for brain surgery.

Other than the ***sleep()***, explicit waits are easy to work with, since Python, Java, and C# bindings all include convenient methods which work with waits for specific expected conditions. In Python, these waits are coded by using the WebDriver method, ***WebDriverWait()***, in conjunction with an

ExpectedCondition. Expected conditions are defined for many common conditions that may occur and are accessed by including the selenium.webdriver.support module as shown below:



```
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
```

Figure 79: Importing Explicit Wait Methods

What this code does is make available to the automator the ability to use any pre-defined expected waits that are available. Calling the explicit wait can be done using the following code (assuming the imports above are made):

wait = WebDriverWait(driver, 10)
element = wait.until(EC.element_to_be_clickable((By.ID, ‘someID’)))

This code will wait up to 10 seconds, checking every 500 milliseconds, for an element to be identified by ***ID*** defined as ‘someid’. If the WebElement is not found after 10 seconds, the code will throw a TimeoutException. If the WebElement is found, a reference to it will be placed in the variable *element* and the code will continue.

Many automators will include this code in wrapper functions so that every element is protected from slow-to-act WebElements.

There are a variety of these expected conditions defined for use in Python, including:

- title_is
- title_contains
- presence_of_element_located
- visibility_of_element_located
- visibility_of
- presence_of_all_elements_located
- text_to_be_present_in_element
- text_to_be_present_in_element_value
- frame_to_be_available_and_switch_to_it
- invisibility_of_element_located
- element_to_be_clickable
- staleness_of
- element_to_be_selected
- element_located_to_be_selected
- element_selection_state_to_be
- element_located_selection_state_to_be
- alert_is_present

Custom wait conditions can also be created, but that is beyond the scope of this syllabus.

4.3 Page Objects

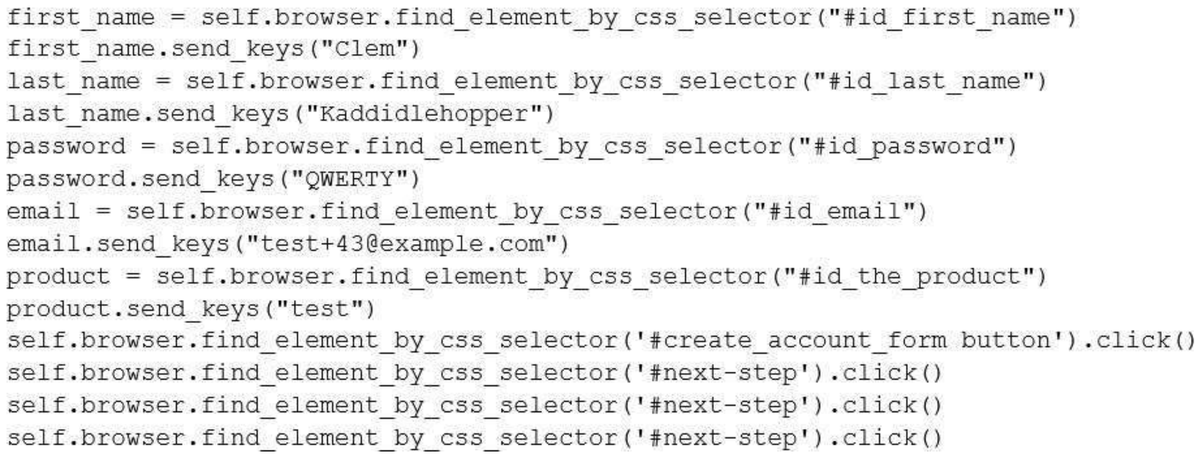
Earlier in the chapter, we recommended that, wherever possible, we should remove complexity from the scripts and place it into the TAA/TAF. We are going to address that topic a bit further in this section when we discuss *Page Objects*, which is representative of a pattern: the *Page Object Pattern.*

A Page Object represents an area in the web application interface with which your test is going to interact. There are three main reasons to use these:

- It creates reusable code that can be shared by multiple test scripts
- It reduces the amount of duplicated code
- It reduces cost and maintenance efforts for test automation scripts
- It encapsulates all operations on the SUT's GUI in one layer
- It gives a clear separation between the business and the technical parts for the test automation design
- When change inevitably occurs, it gives a single point to repair all relevant scripts

Page Objects and Page Object Pattern have been mentioned many times through this syllabus. Page Object Pattern means using Page Objects in the Test Automation Architecture, so these terms may be used almost interchangeably.

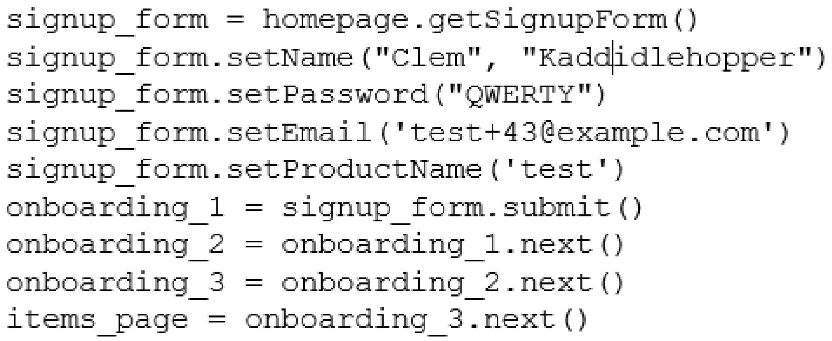
One of the core principles of building a maintainable TAA is dividing it into layers. One of the layers of ISTQB® general TAA (as defined in the TAE syllabus) is the Test Abstraction Layer; this layer takes care of interfacing between the logic of the test case and the physical needs of driving the SUT. Page Objects are part of this layer, usually abstracting the GUI of the SUT. Abstracting here means to hide details of how we are controlling the GUI inside functions that are used in other scripts. Below is a fragment of code that might appear in a script without using the Page Object Pattern. Notice that it is not easily readable; the locators for particular controls are tied intimately with the code.



```
first_name = self.browser.find_element_by_css_selector("#id_first_name")
first_name.send_keys("Clem")
last_name = self.browser.find_element_by_css_selector("#id_last_name")
last_name.send_keys("Kaddidlehopper")
password = self.browser.find_element_by_css_selector("#id_password")
password.send_keys("QWERTY")
email = self.browser.find_element_by_css_selector("#id_email")
email.send_keys("test+43@example.com")
product = self.browser.find_element_by_css_selector("#id_the_product")
product.send_keys("test")
self.browser.find_element_by_css_selector('#create_account_form button').click()
self.browser.find_element_by_css_selector('#next-step').click()
self.browser.find_element_by_css_selector('#next-step').click()
self.browser.find_element_by_css_selector('#next-step').click()
```

Figure 80: Some Nonsense Code Before Page Objects

Now, suppose you take the same code after creating a Page Object:

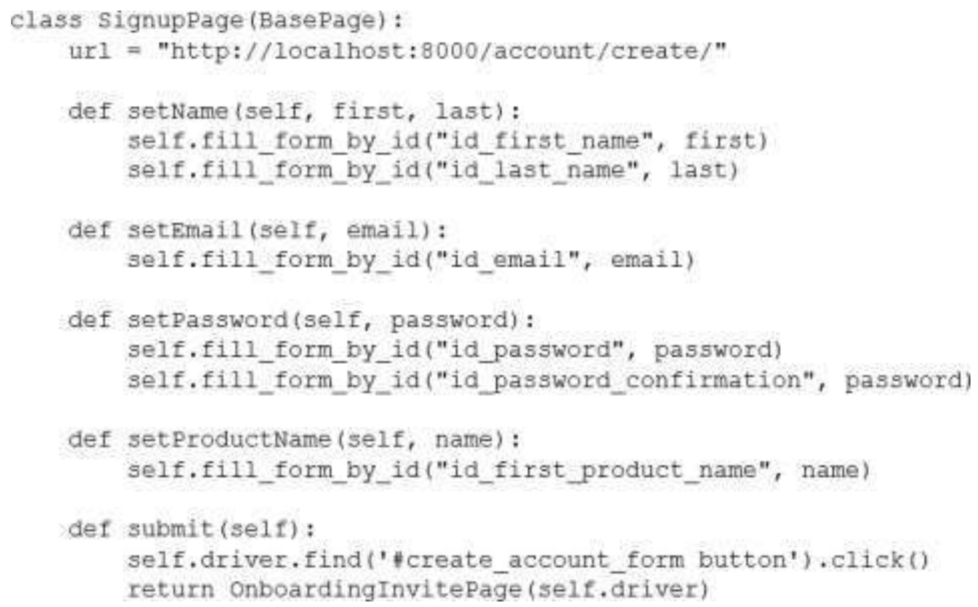


```
signup_form = homepage.getSignupForm()
signup_form.setName("Clem", "Kaddidlehopper")
signup_form.setPassword("QWERTY")
signup_form.setEmail('test+43@example.com')
signup_form.setProductName('test')
onboarding_1 = signup_form.submit()
onboarding_2 = onboarding_1.next()
onboarding_3 = onboarding_2.next()
items_page = onboarding_3.next()
```

Figure 81: Same Code Using Page Object

Quite a bit more readable and succinct.

All of the stuff in the first code snippet still exists. It has just been encapsulated in the Page Object and surfaced as function calls in the second code snippet. The Page Object would look something like this:



```
class SignupPage(BasePage):
    url = "http://localhost:8000/account/create/"

    def setName(self, first, last):
        self.fill_form_by_id("id_first_name", first)
        self.fill_form_by_id("id_last_name", last)

    def setEmail(self, email):
        self.fill_form_by_id("id_email", email)

    def setPassword(self, password):
        self.fill_form_by_id("id_password", password)
        self.fill_form_by_id("id_password_confirmation", password)

    def setProductName(self, name):
        self.fill_form_by_id("id_first_product_name", name)

    def submit(self):
        self.driver.find('#create_account_form button').click()
        return OnboardingInvitePage(self.driver)
```

Figure 82: Example of Page Object

We have abstracted much of the complexity from the script and moved it to the TAA. The complexity is still there, make no mistake about it. The complexity is what allows us to create useful automation and we need it. By hiding it out of sight, however, we can make it easier for the scripters to create valid, powerful scripts quicker.

This hiding of complexity, of dealing with it where and when we want, is a best practice in programming. Consider that many years ago, the first programming was done by plugging in patch cords that directly connected inputs to the processor. Later, all programmers wrote in assembler, crafting code that essentially spoke the same language as the computer processor but not needing physical connections. Then came higher order languages, Fortran, Cobol, C, which were much easier to program; the code was compiled to create the object code that would work with the processor, but the programmer did not need to know the processor microcode. Object oriented languages, C++, Delphi, Java, came and made it even easier.

In this class we use Python which hides most of the complexity of dealing with the browser objects. Using the Page Object Pattern is just one more step.

A Page Object is a class, a module or a set of functions that contains the interface to a form, a page or a fragment of the page of the SUT which a test automator wants to control.

A Page Object exports business operations that are used as test steps in Test Execution Layer. A Page Object Pattern is actually one of implementations of Keyword Driven Testing, which allows a project to lower maintenance effort. We will discuss keyword driven testing in the next section.

The Page Object Pattern can be especially important when maintaining test scripts and can substantially reduce the time needed to update the scripts after changes to the GUI of the SUT.

Beware that Page Object Pattern is not a panacea for troubles with test automation maintenance. While it can lower the costs, in some situations, such as changing the logic of a SUT, updating test scripts can still take much time. Notice that this is not an automation-only issue. If you change the logic of the SUT, your manual test cases would also need to change. Luckily, changing SUT logic does not happen often, because it would force the users of that application to relearn how to use it.

When dividing TAA into layers and designing Page Objects you should observe several general rules:

- Page Objects should not contain business assertions nor verification points
- All assertions and technical verifications regarding the GUI (e.g., checking if a page has finished loading) should be done only within Page Objects
- All waits should be encapsulated in Page Objects
- Only the Page Object should contain calls to Selenium functions
- A Page Object does not need to cover the whole page or form. It may control a section or other specific part of it.

In a well-designed TAA, a Page Object in a Test Abstraction Layer encapsulates calls to Selenium WebDriver methods, so that the Test Execution Layer deals only with the business tier. In such (i.e., well-designed) TAAs there is no import or usage of the Selenium library in the Test Execution Layer.

4.4 Keyword Driven Testing

In chapter one, section two of this syllabus, we discussed the efficacy of a manual test case. We wrote:

> *At its bare minimum, a manual test script tends to have information in three columns. The first column will contain an abstract task to do. Abstract, so it does not need to be changed when the actual software changes. For example, the task "Add Record to Database" is an abstract idea. No matter what version of what database, this abstract task can be performed—assuming a manual tester has the domain knowledge to translate it into concrete actions.*

The idea of an *abstract task* to do is a very powerful idea. Consider the task of opening up a document in a word processor. Let's call it *OpenFile*. We had to open a file in word processors in DOS. Windows 3.1. Windows 95. Macintosh. Unix. Every version of every processor ever made required that task to be done. The 'how' of opening a file has changed for every word processor ever made, but the 'what' has remained the same. *OpenFile* is a prototype for a keyword.

The heart of Keyword Driven Testing (KDT) is this idea that every application has a set of different tasks that need to be done to use the application. We open and save files. We create, read, update, and delete database records. These tasks tend to be an abstract representation of what a user needs to do to interact with the software. In that these are abstract ideas, they rarely change as versions of the software are updated.

Manual tests take advantage of this abstraction; the functional tasks for almost any application rarely change. Once a user learns the interface, the functional tasks that they need to do with the application will almost always remain the same. The way they do the individual tasks may change, but the concepts behind the tasks don't change. Manual tests don't need to be changed every release because the 'what' we test tends to be very stable as compared to the 'how'.

The keywords that we create are essentially a meta-language which we can use for testing. Like manual testers, writing manual test scripts, we identify these 'what' tasks in software and identify them with a descriptive term which becomes our keyword. Test analysts are the ideal people to define the keywords that they need to test an application.

One of the most important prerequisites for implementing maintainable test scripts (manual or automated) is a good structure where the tasks to be performed remain abstract, such that they rarely change. Again, we find that the separation of physical interface from the abstract task is an excellent design technique.

In chapter one, section five, when the Test Automation Architecture was described, we emphasized the well-defined division of test automation into layers. The test case which is described in the test definition layer in abstract terms. The test adaptation layer which interfaces between the test case and the physical GUI of the SUT. The test execution layer which includes the tool that is going to actually run the tests against the SUT.

ISTQB® defines KDT as a scripting technique that uses data files to contain not only test data and expected results, but also keywords (these abstract statements of 'what' the system is supposed to do). Keywords are often also called *Action Words*. The ISTQB® definition also says that the keywords are interpreted by special supporting scripts that are called by the control script for the test.

Let's look at this definition more closely. KDT is a scripting technique, which means that it is a way of automating tests. KDT files contain test data, expected results and keywords.

Keywords are business actions or steps of a test case. These are exactly the same as column one in a manual test case.

When implementing test automation with KDT principles, the test cases, themselves, do not contain specific actions to be taken on the SUT (the 'how' of the action). The automated test cases contain sequences of test steps which are usually abstract, high level business actions (e.g., "login to a system", "make a payment", "find a product"). The exact, physical actions on the SUT are hidden within the implementation of the keywords (e.g., in the Page Objects as described in the previous section). Note that this duplicates the separation between a manual test case and the human tester who is going to add context and reasonableness to the test while physically manipulating the GUI to run the test.

This approach has several advantages:

- The test case design is decoupled from the SUT
- A clear distinction is made between the Test Execution, the Test Abstraction, and the Test Definition layers
- An almost complete division of labor:
    - Test Analysts design test cases and write scripts using keywords, data, and expected results
    - Technical Test Analysts (automators) implement keywords and the execution framework needed to run the tests
- Reuse of keywords in different test cases
- Improved readability of test cases (they look pretty much like manual test cases)
- Less redundancy
- Lower maintenance costs and effort
- If the automation is off line, a manual tester can execute the test manually directly off the automated script
- Because keyword tests are abstract, the scripts using keywords can be written well before the SUT is available for testing (just like manual tests)
- The previous bullet means that automation can be ready earlier and pushed into functional testing and not just regression testing
- A few technical automators can support an unlimited number of test analysts making the architecture totally scalable
- Because the scripts are completely separated from the implementation level, different tools may be used interchangeably

Sometimes libraries of keywords use other keywords, so the whole keyword structure can become quite complex. In such a structure some keywords will have high level of abstraction e.g., “Add a product X to the invoice with price Y and quantity Z”, and some of them will be quite low level e.g., “Click >>Cancel<< button”.

Therefore, keywords have several places of definition and usage in the general Test Automation Architecture:

- Low level keywords are implemented as Page Objects in the Test Adaptation Layer, they do the actions on the SUT
- Low level keywords are used as parts of higher level keywords in the Test Library of the Test Definition Layer
- High level keywords are implemented in the Test Library
- High level keywords are used as test steps of Test Procedures in Test Execution Layer

The ISTQB® definition of KDT says that keywords are interpreted by special supporting scripts. That is true when KDT is implemented using specialized tools such as Cucumber or RobotFramework.

However, it is possible to observe KDT principles while implementing a TAF with general purpose programming languages such as Python, Java or C#. In this case each keyword becomes the invocation of a function or method of the objects of the Test Library.

There are two ways of implementing KDT in practice. The first, classical KDT, as described by Dorothy Graham in her test automation book[2], is a top-down approach. Its first step is to design test cases as they would be designed for manual tests. Then the steps of these test cases are abstracted to become high level keywords, which may be decomposed into low level keywords or are implemented in a tool or a programming language. This method is most cost-effective when the Test Library already contains many functions/methods which perform tasks against the SUT. It can also be useful when automated scripts are converted from manual test cases rather than being written from scratch.

The second way is the implementation of scripts in a bottom-up direction. This means that scripts are recorded by a tool (e.g., Selenium IDE) and then are refactored and restructured into a proper KDT test automation architecture. This approach allows test teams to quickly write several test automation scripts and run them against the SUT.

Unfortunately, writing more than twenty or thirty scripts using this quick-and-dirty approach without refactoring them to be a well-designed TAF will expose the automation project to a risk of future high maintenance efforts and costs. In addition, tests that are first scripted without following manual test cases stand the risk of not being good tests (i.e., they may not actually mitigate important risks that need to be tested).

When implementing TAF basing on KDT principles consider also the following aspects:

- Fine-grained keywords allow more specific scenarios, but at the cost of script maintenance complexity
- The more fine-grained a keyword is, the more likely it is tied intimately to the interface of the SUT and the less abstract it is (e.g., the example given earlier, “Click >>Cancel<< button”
- Initial keyword design is important, but new and different keywords will ultimately be needed, which implicates both business logic and the automation functionality to execute it

When done correctly, KDT has been shown to be a good automation approach which can produce scripts with consistently low maintenance costs. It allows building well-structured test automation frameworks, and uses best practices of software design.

[2] “Software Test Automation”, Mark Fewster and Dorothy Graham, Addison-Wesley Professional, 1999

Appendix - Glossary of Selenium Terms

class attribute: An HTML attribute that points to a class in a CSS style sheet. It can also be used by a JavaScript to make changes to HTML elements with a specified class

comparator: A tool to automate the comparison of expected results against the actual results

CSS selector: Selectors are patterns that target the HTML elements you want to style

Document Object Model (DOM): An application programming interface that treats an HTML or XML document as a tree structure wherein each node is an object representing a part of the document

fixture: A test fixture is a mock object or environment used to consistently test some item, device, or piece of software

framework: Provides an environment for automated test scripts to execute; including tools, libraries, and fixtures

function: A Python function is a group of reusable statements that perform a specific task.

hook: An interface which is introduced into a system which is created predominately to provide enhanced testability to that system

HTML (HyperText Markup Language): The standard markup language for creating web pages and web applications

ID: An attribute that specifies a unique identification string for an HTML element. The value must be unique within the HTML document

iframe: An HTML inline frame, used to embed another document within an HTML document

modal dialog: A screen or box which forces the user to interact with it before they can access the underlying screen

Page Object Pattern: A test automation pattern which requires that technical logic and business logic be dealt with at different levels

pesticide paradox: A phenomenon where repeating the same test multiple times causes it to find fewer defects

persona: A user profile created to represent a user type that interacts with a system in a common way

pytest: A Python testing framework

tag: HTML elements are delineated by tags, written using angle brackets

technical debt: Implies the additional cost of rework caused by choosing to ignore bad designs or implementations in the short term

WebDriver: The interface against which Selenium tests are written. Different browsers can be controlled via different java classes, e.g., ChromeDriver, FirefoxDriver, etc.

wrapper: A function in a software library whose main purpose is to call another function, often adding or enhancing functionality while hiding complexity

XML (eXtensible Markup Language): A markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable

XPath (XML Path Language): A query language for selecting nodes from an XML document