Lec7

Various XPath functions are used for selecting elements based on text, attributes, or position within an HTML document. Here's a brief explanation of each concept:

1. `text()` Function:

- Purpose: Used to find elements that contain or match specific text.
- Example:
  - `//p[text()='PracticeAutomationHere']`: Selects a `<p>` tag that exactly matches the text "PracticeAutomationHere".
- Usage: It matches elements that contain exact text, useful when the visible text of an element needs to be located.

2. `contains()` Function:

- Purpose: Finds elements that have a partial match within an attribute or text.
- Example:
  - `//input[contains(@value, 'ra')]`: Selects `<input>` tags whose `value` attribute contains "ra".
  - `//p[contains(text(), 'Automation')]`: Finds a `<p>` tag containing the partial text "Automation".
- Usage: Ideal when attribute values or text change dynamically, or you only know part of the string.

3. `starts-with()` Function:

- Purpose: Selects elements based on the starting characters of an attribute value or text.
- Example:
  - `//input[starts-with(@value, 'o')]`: Finds input elements where the value starts with "o".
  - `//p[starts-with(text(), 'Practice')]`: Selects `<p>` tags that start with the text "Practice".
- Usage: Used when you want to match elements whose attribute or text begins with specific characters, useful for partial or dynamically generated values.

4. `last()` Function:

- Purpose: Selects the last element in a set of matching nodes.
- Example:
  - `//body/*[last()]`: Selects the last child element of the `<body>` tag.
  - `//p[last()]`: Finds the last `<p>` tag.
  - `//p[last()-1]`: Finds the second-to-last `<p>` tag.
- Usage: Useful for locating the last or the last-but-one element in a list of nodes, such as the last paragraph or input field.

5. `position()` Function:

- Purpose: Finds an element at a specific position in a node set.
- Example:
    - `//p[position()=1]`: Selects the first `<p>` tag.
    - `//input[position()=8]`: Selects the 8th `<input>` tag.
- Usage: Used for targeting elements at particular positions in a collection of nodes. Perfect when you know the exact index or order of an element.

Summary of Key Functions:

- `text()`: Exact text matching.
- `contains()`: Partial matching for attributes or text.
- `starts-with()`: Matching the beginning of an attribute or text.
- `last()`: Selects the last element in a set.
- `position()`: Finds elements by their position in the document.

These functions help with dynamically locating web elements that have changing attributes or texts, and when you want to match based on partial or positional criteria.

Lec-8

This provides an overview of XPath Axes and Types. These axes and types allow you to navigate through XML or HTML documents based on relationships between elements. Here's a breakdown of the concepts:

XPath Axes:

1. `following` Axis:
   - Purpose: Selects everything in the document after the closing tag of the current node.
   - Examples:
     - `//head/following::body`: Selects the `<body>` tag that comes after the `<head>` tag.
     - `//body/div[1]/div/following::div`: Selects all the `<div>` tags that come after a particular `<div>` inside `<body>`.
   - Usage: Use this when you need to select elements that come after a specific tag.
2. `preceding` Axis:
   - Purpose: Selects all nodes that appear before the current node in the document, except parent and ancestor nodes.
   - Examples:
     - `//body/preceding::head`: Selects the `<head>` tag that comes before the `<body>` tag.
     - `//body/div[4]/preceding::div`: Finds all `<div>` tags before a specific `<div>`.
   - Usage: Useful for selecting elements that come before the current node.
3. `following-sibling` Axis:
   - Purpose: Selects all sibling elements after the current node.
   - Examples:
     - `//body/div[1]/following-sibling::div`: Finds all sibling `<div>` tags after the first `<div>` in the `<body>`.
     - `//p[1]/following-sibling::p`: Selects all `<p>` tags that are siblings and follow the first `<p>`.
   - Usage: Use this axis to find nodes that are at the same hierarchical level but come after the current element.
4. `preceding-sibling` Axis:
   - Purpose: Selects all siblings before the current node.
   - Examples:
     - `//body/div[4]/preceding-sibling::div`: Finds all sibling `<div>` tags before the fourth `<div>` in the `<body>`.

        - `//p[2]/preceding-sibling::p`: Selects all `<p>` tags that are siblings before the second `<p>`.
    - Usage: This is helpful when you want to locate sibling nodes before the current element.

XPath Types:

1. `parent` Axis:
    - Purpose: Selects the parent of the current node.
    - Examples:
        - `//head/parent::html`: Finds the parent of the `<head>` tag, which is `<html>`.
        - `//body/parent::html`: Selects the parent of `<body>`, which is also `<html>`.
    - Usage: When you need to traverse upwards and find the parent element of a specific node.
2. `child` Axis:
    - Purpose: Selects all child elements of the current node.
    - Examples:
        - `//html/child::head`: Selects the child `<head>` tag of the `<html>` tag.
        - `//body/child::div[1]`: Selects the first child `<div>` inside the `<body>` tag.
    - Usage: Helps when you want to find direct child elements of a node.
3. `ancestor` Axis:
    - Purpose: Selects all ancestors (parent, grandparent, etc.) of the current node.
    - Examples:
        - `//title/ancestor::html`: Finds the ancestor `<html>` tag for the `<title>` element.
        - `//head/ancestor::html`: Finds the ancestor `<html>` tag for the `<head>` element.
    - Usage: Use this to trace up through the hierarchy and find any ancestor node.
4. `descendant` Axis:
    - Purpose: Selects all descendants (children, grandchildren, etc.) of the current node.
    - Examples:
        - `//html/descendant::title`: Finds the `<title>` element, which is a descendant of `<html>`.
        - `//html/descendant::body`: Finds the `<body>` element, which is also a descendant of `<html>`.
    - Usage: Useful when you want to find all nested child elements under a specific node.

Summary of XPath Axes and Types:

- Axes like `following`, `preceding`, `following-sibling`, and `preceding-sibling` allow you to navigate between elements that have a specific relationship in terms of order or hierarchy.
- Types like `parent`, `child`, `ancestor`, and `descendant` help in navigating through parent-child or ancestor-descendant relationships.

These XPath techniques provide flexibility to locate and traverse through complex XML or HTML documents even when IDs, names, or classes aren't available.

Lec-9

This provides a detailed breakdown of CSS Selectors and how they differ between absolute and relative paths. Here's an explanation of the key concepts:

1. CSS Selectors vs XPath Expressions

- CSS Selectors are described as being slightly faster than XPath expressions when selecting elements from a web page's HTML document.

2. Types of CSS Selectors

- Absolute (Complete Path): These selectors locate elements by specifying the complete path starting from the root (usually `html`) to the target element. It is a complete reference of the hierarchy.
- Relative (Direct/Shortcut): These selectors locate elements directly without referencing the full hierarchy from the root. They are more efficient and easier to maintain.

3. Absolute CSS Selectors

- Purpose: Absolute selectors try to locate the element by referencing the complete path starting from the root element (`html`).
- Examples:
  - `html` - Locates the entire HTML document.
  - `html > head` - Locates the `head` portion of the HTML.
  - `html > body` - Locates the `body` portion of the HTML.
  - `html > body > p` - Locates all `p` tags inside the `body` section.
  - `html > body > p[id='para1']` - Locates a `p` tag with `id='para1'`.
  - `html > body > p[class='sub']` - Locates a `p` tag with the class `sub`.
  - `html > body > p#para1` - Locates the `p` tag with id `para1`.
  - `html > body > p.sub` - Locates a `p` tag with the class `sub`.
  - `html > body > p[id='para1'][class='main']` - Locates the `p` tag with both `id='para1'` and `class='main'`.

Tools Limitation: SelectorsHub cannot auto-generate absolute CSS selectors.

Disadvantage: Absolute CSS selectors are rigid because they depend on the entire hierarchy of elements from the root. If the structure changes even slightly, the selectors can break.

4. Relative CSS Selectors

- **Purpose: These selectors directly locate elements without starting from the root, making them more adaptable to changes in the HTML structure.**
- **Examples:**
  - `html` - Locates the HTML tag.
  - `head` - Locates the `head` portion of the HTML.
  - `title` - Locates the `title` portion inside the `head`.
  - `body` - Locates the body portion of the HTML.
  - `p` - Locates all `p` tags in the body.
  - `p[id='para1']` - Locates a `p` tag with id `para1`.
  - `p[class='sub']` - Locates a `p` tag with the class `sub`.
  - `p[id='para1'][class='main']` - Locates a `p` tag with both `id='para1'` and `class='main'`.
  - `p#para1` - Locates a `p` tag with the id `para1`.
  - `p.sub` - Locates a `p` tag with the class `sub`.

Tool Support: SelectorsHub can auto-generate relative CSS selectors.

Advantages of Relative CSS Selectors:

- **They are more flexible and adaptable to changes in the HTML structure since they don't rely on the full path.**
- **Easier to maintain, especially in dynamic pages where the structure might change.**

5. Summary:

- **Absolute CSS selectors use the full hierarchy and are prone to breaking if the structure changes.**
- **Relative CSS selectors are more flexible and are often preferred for their ease of use and efficiency.**

Lec-10

This expands on various CSS selector concepts and examples, explaining different techniques for selecting HTML elements using CSS selectors. Here's a detailed breakdown of each section in the image:

1. Basic HTML Structure

- `HTML page > html`: Locates the complete `html` element of the page.
- `HTML head > head`: Locates the `head` section of the HTML.
- `HTML title > title`: Locates the `title` tag within the `head` section.
- `HTML body > body`: Locates the `body` section of the HTML.
- `p tag > p`: Locates all `p` tags within the HTML document.

2. Tag Inside Another Tag

- `p tags inside body`: `body > p`: Locates all `p` tags inside the `body`.
- `p tags inside html`: `html > p`: Locates all `p` tags directly inside the `html` element.

3. Locate Elements by Attribute

- Locate p tag having id "para1": `p[id='para1']`: This selects the `p` tag with an `id` of `para1`.
- Locate a tag with class "main": `p[class='main']`: This selects a `p` tag with the `class="main"`.
- Locate elements with id "para1": `[id='para1']`: This selects any element (not just `p` tags) that has the `id="para1"`.
- Locate elements with class "sub": `[class='sub']`: This selects any element with the `class="sub"`.

4. Using Combinations of Attributes

- Using # for locating elements by id:
  - `p#para1`: Locates the `p` tag with `id="para1"`.
  - `p.tag1#para2`: Locates a `p` tag with the class `tag1` and `id="para2"`.
- Using . for locating elements by class:
  - `p.sub`: Locates a `p` tag with the class `sub`.
  - `p.tag1.sub`: Locates a `p` tag with the classes `tag1` and `sub`.
- Locate elements having class "main": `.main`: Locates elements with the class `main`.
  - For example, `p.main`: Locates `p` tags with class `main`.

5. Advanced Attribute Selectors

- Locate input with type attribute having value "text": `input[type='text']`: This selects an `input` tag with a `type` attribute of `text`.

- Locate element with attribute value starting with "b": `[value^='b']`: Selects elements with a `value` attribute that starts with "b".
- Locate element with attribute value ending with "lue": `[value$='lue']`: Selects elements with a `value` attribute that ends with "lue".
- Locate element with attribute value containing "min": `[value*='min']`: Selects elements with a `value` attribute that contains "min".

6. Pseudo-Classes and Pseudo-Elements

- Locate the first child:
  - Of body: `body > p:first-child`: Locates the first `p` tag that is the direct child of `body`.
  - Of html: `html > *:first-child`: Locates the first child element of the `html` tag.
- Locate the last child:
  - Of body: `body > p:last-child`: Locates the last `p` tag that is the direct child of `body`.
  - Of html: `html > *:last-child`: Locates the last child element of the `html` tag.
- Locate the nth child:
  - Second child in body: `body > *:nth-child(2)`: Locates the second child of the body tag.
  - First child inside html: `html > *:nth-child(1)`: Locates the first child of the `html` tag.

7. Other Advanced Selectors

- Locate first child with a specific class: `p#para1:first-child`: Locates the first child `p` tag with `id='para1'`.
- Locate the second child of a class with specific id: `p#para3.para:nth-child(2)`: Locates the second child with `id='para3'` and class `para`.

8. Grouping and Logical Operations in CSS Selectors

- Using `,` for grouping selectors:
  - `p[class="a"], body, ul`: This will highlight all `p` tags with `class="a"`, the `body`, and `ul` tags.
- Using `[attr]` for matching any attribute:
  - `p[class]`: This selects all `p` tags that have a `class` attribute.
- Using logical AND and OR in selectors:
  - `p[id="para1"].main`: This selects a `p` tag that has both `id="para1"` and the `class="main"`.
  - `p[id="para1"][class="main"]`: Another way to do the same thing, selecting a `p` tag with both `id="para1"` and `class="main"`.

9. Specific Element Selection

- Selecting parent-child relationships:

    - `p#id > a`: Locates `a` tags that are direct children of `p` tags with the `id`.
    - Locate following siblings: `p + ul`: Locates `ul` elements that directly follow `p` tags.

10. Other Miscellaneous Selectors

- Locate disabled elements: `input:disabled`: Selects any `input` element that is disabled.
- Locate enabled elements: `input:enabled`: Selects any `input` element that is enabled.
- Locate checked checkboxes: `input:checked`: Selects any checkbox `input` that is checked.
- Locate selected dropdown options: `select option:checked`: Locates the selected option in a dropdown field.

Conclusion:

This image provides an in-depth understanding of CSS selectors, ranging from basic tag and attribute selections to advanced pseudo-classes and attribute-based logic. These selectors enable more precise and complex ways to target elements in a webpage's DOM for styling or manipulation through JavaScript and CSS.

Lec-11

This provides more advanced examples of Relative CSS Selectors and showcases their advantages in locating HTML elements efficiently. Here's an explanation of the concepts:

Advantages of Relative CSS Selectors

Relative CSS selectors allow you to locate elements directly without having to specify the entire path from the root element. This makes them more flexible and adaptable to changes in the HTML structure.

Examples of Relative CSS Selectors

- HTML page: Targets the entire `html` document.
- HTML Head: Targets the `head` section of the document.
- HTML Title: Targets the `title` tag within the `head` section.
- HTML Body: Targets the `body` section of the HTML.

Selecting Tags and Elements

- p tags: Selects all `p` tags in the document.
- p tags inside body: Targets `p` tags that are direct children of the `body`.
- p tags inside html: Targets `p` tags that are direct children of the `html` element.

Locating Elements by Attribute

- Locate p tag having id "para2": `p[id='para2']` locates a `p` tag with the `id="para2"`.
- Locate p tag having class "main": `p[class='main']` locates a `p` tag with the `class="main"`.
- Locate elements having id "para1": `[id='para1']` selects any element with the `id="para1"`.
- Locate elements having class "sub": `[class='sub']` selects any element with the `class="sub"`.

Using Selectors for IDs and Classes

- Using # for locating elements by id:
    - `#para1` selects the element with the `id="para1"`.
- Using . for locating elements by class:
    - `.main` selects elements with the class `main`.

Locating Elements by Attribute Values

- Locate input tag having value="blue": `input[value='blue']` locates an `input` element with a `value` attribute equal to "blue".
- Locate elements having value="blue": `[value='blue']` selects elements with the attribute `value="blue"`.

Selecting All Elements of a Tag Type

- Locate all input tags: `input` selects all `input` elements in the document.

Locate Elements by Common Attributes

- Locate elements having 'value' as an attribute: `[value]` selects elements that have the `value` attribute.
- Locate elements having 'id' as an attribute: `[id]` selects elements with the `id` attribute.
- Locate elements having 'name' as an attribute: `[name]` selects elements with the `name` attribute.
- Locate elements having 'href' as an attribute: `[href]` selects elements with the `href` attribute (usually anchor tags `<a>`).
- Locate elements having 'src' as an attribute: `[src]` selects elements with the `src` attribute (such as images).
- Locate img tags having 'src' as an attribute: `img[src]` specifically selects `img` tags with the `src` attribute.

Select Based on Attributes for Specific Tags

- Locate all the p tags having 'id' as attribute: `p[id]` selects all `p` tags that have an `id` attribute.
- Locate all elements having 'class' as an attribute: `[class]` selects elements with the `class` attribute.

Pseudo-Class Selectors

- `:first-child`: Selects the first child of its parent.
- `:last-child`: Selects the last child of its parent.
- `:nth-child(n)`: Selects the nth child of its parent. For example, `p:nth-child(2)` selects the second `p` tag among its siblings.

Combined Selectors

- textarea[id='ta1'], button[id='but2']: Combines selectors using the `,` operator. This selects both the `textarea` with `id='ta1'` and the `button` with `id='but2'`.

Wildcard Selector

- `*`: Selects all elements in the document.
- `head > *`: Selects all child elements of the `head` tag.
- `body > *`: Selects all child elements of the `body` tag.

Attribute Matching Selectors

- `p[class^='ma']`: Selects `p` tags where the class attribute starts with "ma".
- `p[class$='ub']`: Selects `p` tags where the class attribute ends with "ub".
- `p[class*='ai']`: Selects `p` tags where the class attribute contains "ai".

Logical Operators in Selectors

- `p[id='para1'][class='main']`: Combines conditions with logical AND, selecting `p` tags with both `id='para1'` and `class='main'`.
- `p:not([id='para1'])`: Selects `p` tags that do not have `id="para1"`.
- `p:not([id='para1'])[class='sub']`: Selects `p` tags that do not have `id='para1'` but have the class `sub`.
- `p:not([id='para1']):not([class='main'])`: Selects `p` tags that have neither `id='para1'` nor `class='main'`.

Following Sibling Selectors

- `p[id='para1'] + p`: Selects the `p` tag that immediately follows a `p` tag with `id='para1'`.
- `head + *`: Selects any element immediately following the `head` tag.

State Selectors

- Locate disabled elements: `*:disabled` selects all elements that are disabled.
- Locate enabled elements: `*:enabled` selects all elements that are enabled.

Checked and Selected Options

- Locate selected checkbox or radio options: `*:checked` selects all checked or selected options in checkboxes, radio buttons, or drop-down fields.

Lec-12

1. What is Testing?

- Testing is the process of evaluating a system or its components with the intent to find whether it satisfies the specified requirements or to identify any defects. Testing ensures the product works as expected and delivers quality results to the end-users.

2. What is Software?

- Software refers to a set of instructions, data, or programs used to operate computers and execute specific tasks. It is the opposite of hardware, which describes the physical aspects of a computer. Software can be categorized into system software (such as operating systems) and application software (like word processors, games, and more).

3. What is Software Testing?

- Software Testing is a process of executing software or an application to verify if it behaves as expected. The main goal of software testing is to ensure that the software meets the requirements and is defect-free. It involves various levels of testing like unit testing, integration testing, system testing, and acceptance testing.

4. How Testing is Performed Manually?

- Manual Testing is the process of manually executing test cases without using any automation tools. Testers manually interact with the application, check for bugs, and ensure that everything is functioning correctly. It is a time-consuming process, but it can be essential for exploratory, usability, and ad-hoc testing.

5. Why is Automation Testing Required?

- Automation Testing is required to enhance the speed, accuracy, and efficiency of the testing process. Instead of executing repetitive test cases manually, automation testing uses specialized tools to run those tests automatically. This reduces human error, increases coverage, and allows tests to be executed faster across different environments.

Automation testing is particularly useful when there are frequent updates or changes to the application that need to be tested repeatedly.