**CSIT128 - Assignment 1**

**HTML Start and Tab Title: 1**

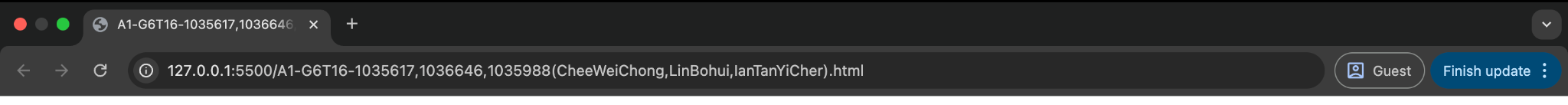
[**Part 1:**](#_8h2xzgbzbmmf) **2**

[**Part 2:**](#_ywyz5sui5r3t) **4**

[**Part 3:**](#_xt9i4krtxpyn) **8**

[**Part 4:**](#_i4p4qbg00clv) **10**

# HTML Start and Tab Title :

****

****

This section defines the beginning of the HTML document and provides metadata for the browser. <html> is the document root; <head> groups non‑visible information such as stylesheets, scripts, and the <title>. The <title> sets the text shown in the browser tab, bookmarks.

# 

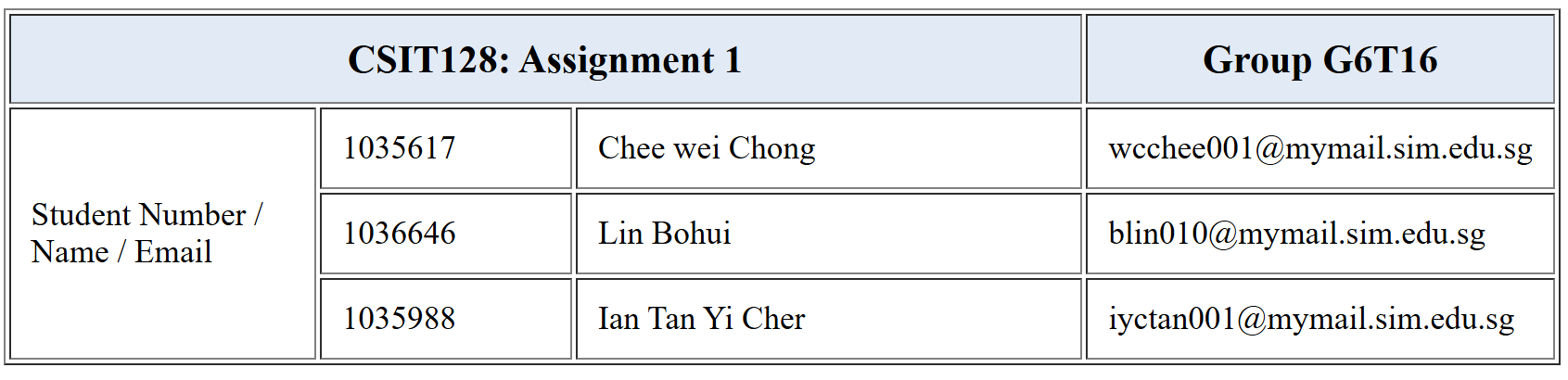
# 

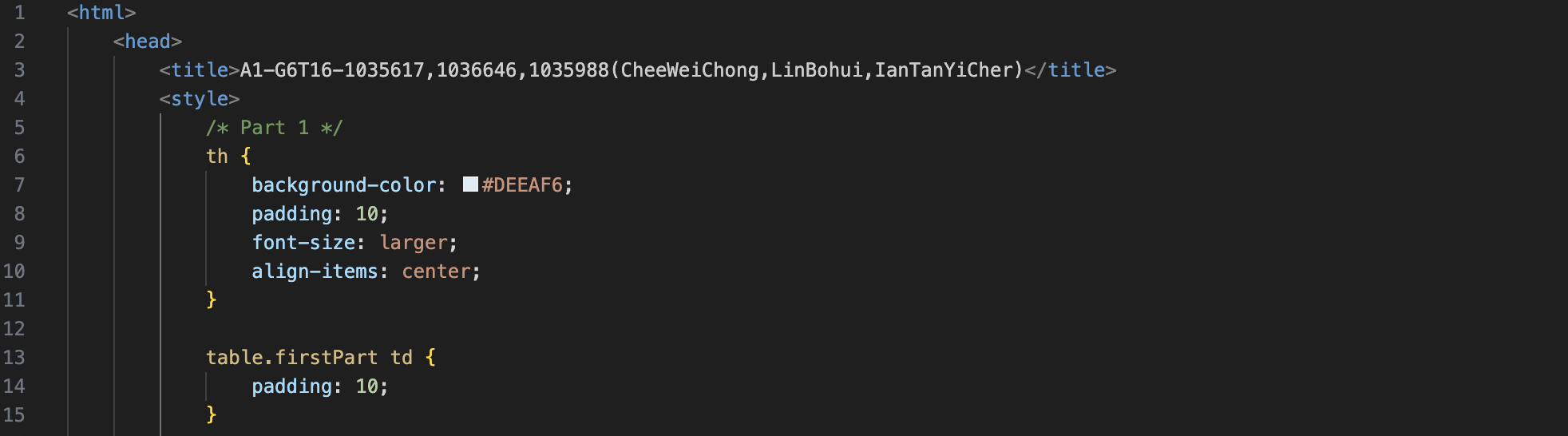
# 

# 

# 

# Part 1:

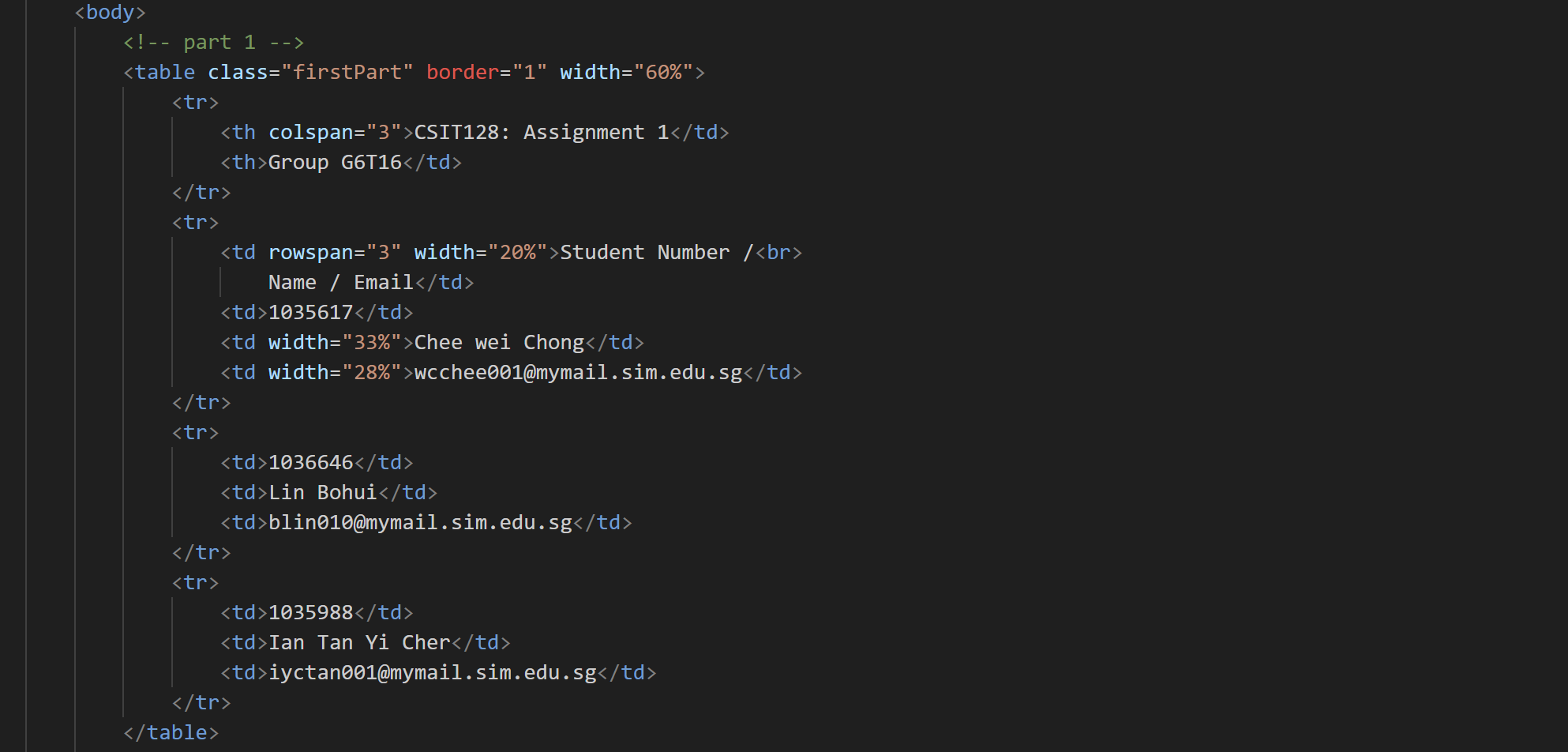




This section defines the internal CSS styles for the table in Part 1. The styling is written inside the <style> tag in the <head> section of the HTML document.

The th selector applies styles to all table header cells. A background color of #DEEAF6 (a light grey-blue) is used to match the formatting shown in the assignment guide. We added padding: 10px; to create consistent spacing inside each header cell, improving readability. The font-size is set to larger to make the header stand out compared to regular table cells.

The class firstPart is applied to the <table> element. The CSS rule table.firstPart td adds padding: 10px; to all table data cells (<td>) inside this table, enhancing clarity and spacing.



This section contains the HTML that defines the structure and content of the table. The table is placed inside the <body> section using the <table> tag. We apply the firstPart class to this table, which gives each data cell (<td>) a padding of 10px for better readability.

The border="1" attribute creates a simple one-pixel border around the table, and width="60%" sets the table’s width to 60% of the page.

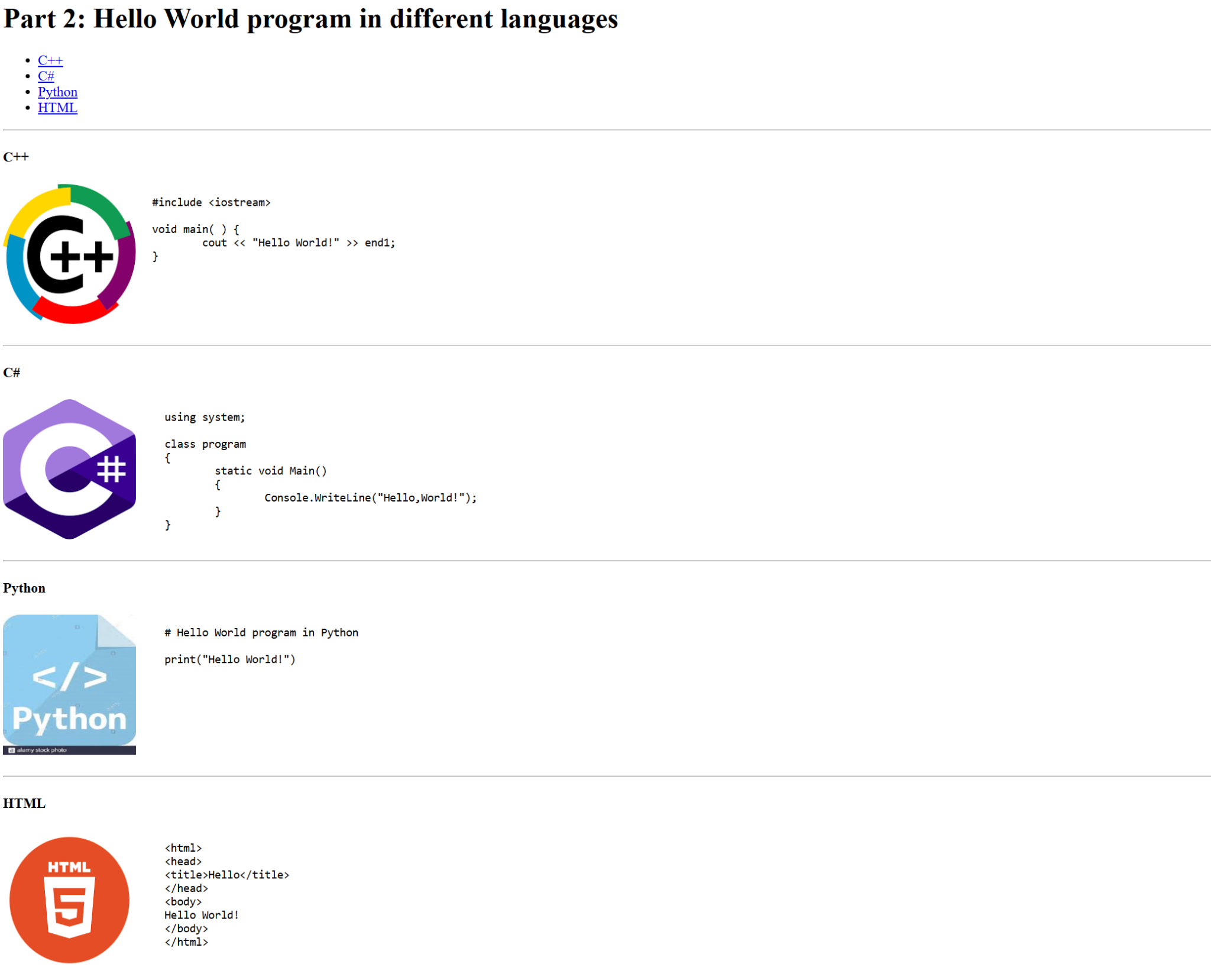
Each row of the table is represented by a <tr> element. The table is made up of 4x4 dimensions where the top row uses two header cells (<th>). The first header uses colspan="3" to span across three columns and display the assignment title. The second header displays the group name. The use of (<th>) for the header improves accessibility as readers can easily discern which module and assignment our report is for, as well as the contributors to the report.

The next row uses rowspan="3" in its first cell to indicate that this cell stretches vertically across three rows, labeling the student information columns required for each group member as stated. Each subsequent cell in the row contains data such as student ID, name, and email. The following two rows repeat the same column structure for the remaining groupmate’s respective information.

Finally, the table is closed with a </table> tag to indicate the end of the table structure.

# 

# Part 2:

****

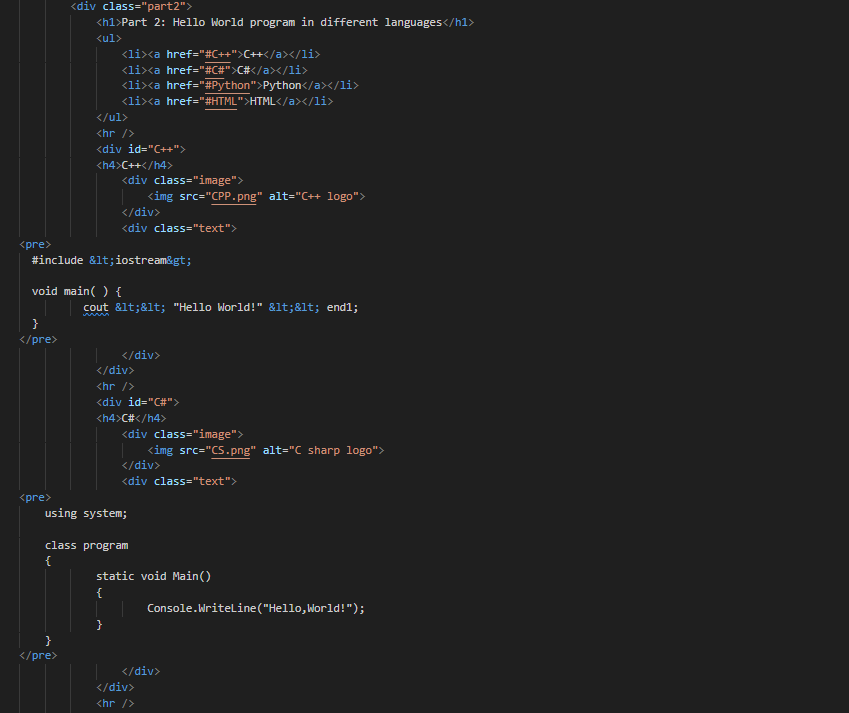


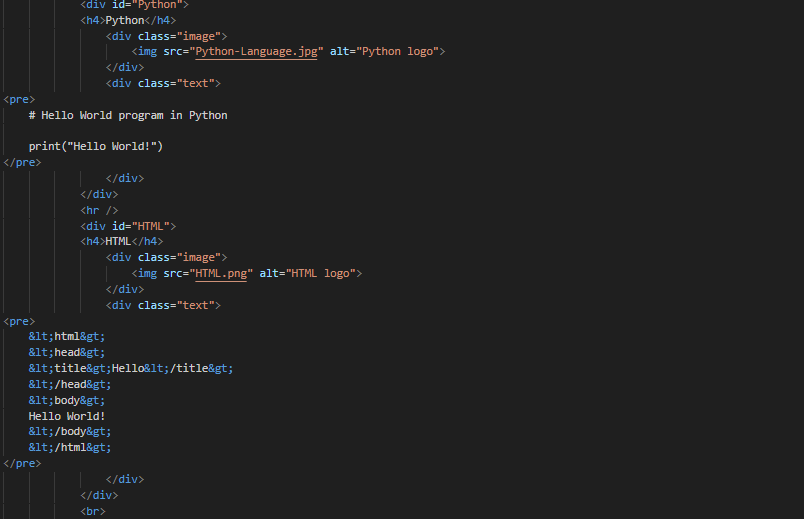
The section above defines the internal CSS styles for Part 2 that is written inside the <style> tag in the <head> section of the HTML document.

img{ } styles all the <img> tags, setting each image's height and width to 10em and 9.5em respectively. ‘The unit em is relative to the font size of the element, where 1em = 16px. Therefore, 10em = 160px and 9.5em = 152px, meaning all images will be 160px tall and 152px wide.

.image{ } styles element with image as its class name (class = “image”), by applying display: inline-block and padding-bottom: 1em. By setting the display to inline-block, it allows adjustment to the height,width,padding and margins of the element while flowing inline with other elements such as texts, making the layout look neater. Padding-bottom adds space below the element/image, creating separation between the image and the content beneath it

.

.text{ } styles elements with text as its class name (class = “text”), by applying display: inline-block and vertical-align: top. By setting display to inline-block, it allows the element to flow inline with another element, in this case, an image. Vertical align: top aligns the top of the element with the top of an adjacent inline-block element,in this case, an image, ensuring the layout looks tidy and properly aligned.



The section above defines the structure and layout for Part 2. The code for part 2 is written inside the <body> section of the HTML document.

Part 2 begins with a heading <h1>. Below the heading, unordered list <ul> is created, containing four list items<li>: C++, C#, Python and HTML. Each <li> has an anchor link <a href=”#...”> that links directly to its corresponding section using the names of the list items as id <div id=”...”>. When a user clicks on any of the four list items, they are brought to the respective section. A horizontal rule <hr/> is added to visually separate content

.

The first section is C++, defined using <div id=”C++”>, followed by a heading <h4>. Inside this section, a class named image <div class=”image”> is created to style the image. Within it, an image of C++ logo is inserted <img src=”CPP.png” alt=”C++ logo”>. Next, a <div class=”text”> is created to style the content in this <div>. In the same <div>, a preformatted text tag <pre> is used to display the text exactly written in the HTML file. Within <pre> a simple C++ code is written. The symbol **&lt;&lt;** are special characters in HTML which represent the characters **<<**. A horizontal line <hr/> follows to separate this section from the next.

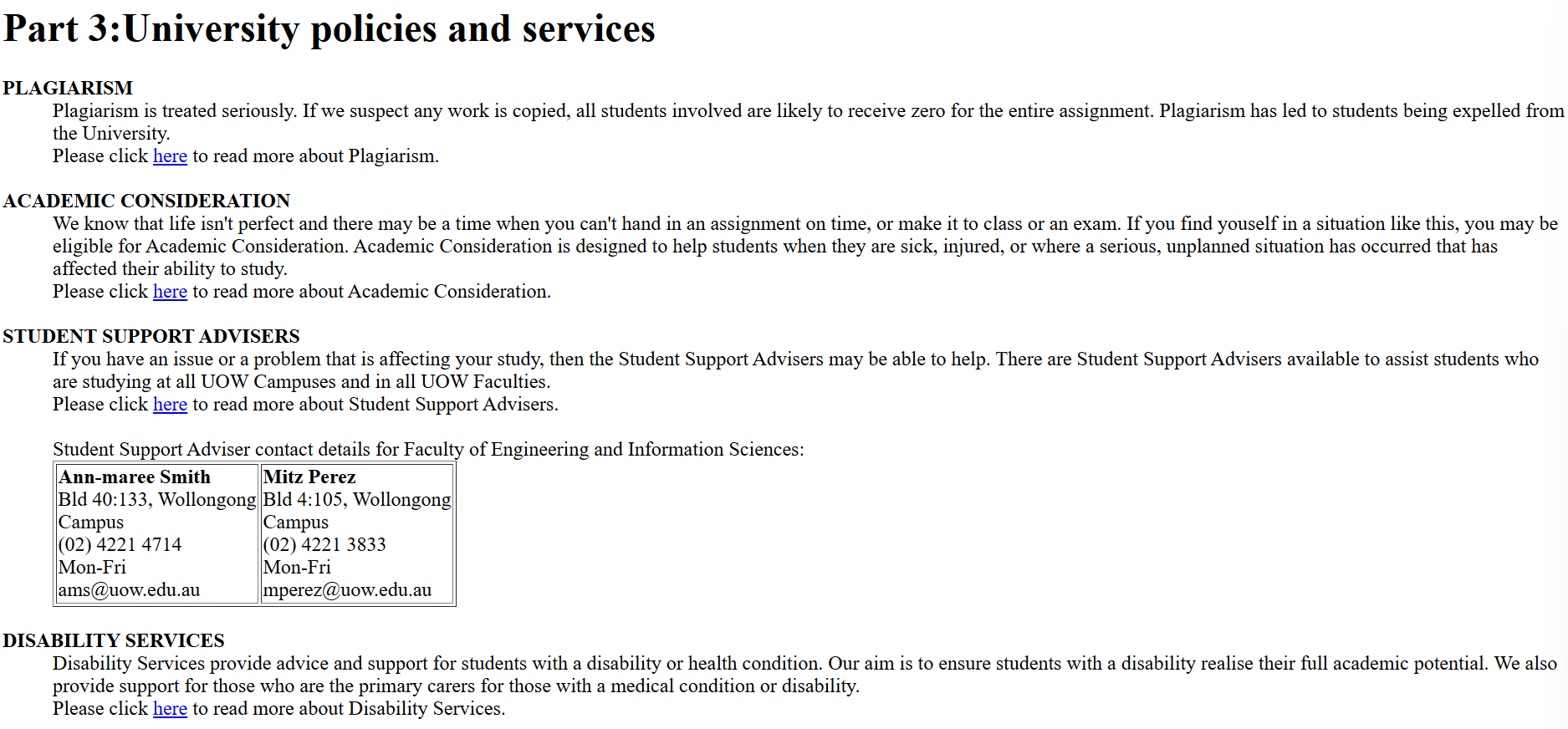
The second section, C#, follows the same structure as C++ section. It starts with <div id=”C#”>, followed by a heading <h4>. A class named image <div class=”image”> is created to style the image. Within it, an image of C# is inserted <img src=”CS.png” alt=”C sharp logo”>. A class named text is created <div class=”text”> contains a preformatted text tag <pre> displaying a simple C# code example. A horizontal line <hr/> separates this section from the next.

The third section, Python, follows the same layout as C++ and C#. It starts with <div id=”Python”>, followed by a heading <h4>. A class named image <div class=”image”> is created. Within it, an image of Python is inserted <img src=”Python-Language.jpg” alt=”Python logo”>. A class named text is created <div class=”text”> contains a preformatted text tag <pre> displaying a simple Python code example. A horizontal line <hr/> is used to separate this section from the next as well.

The fourth section, HTML, follows the same structure. It starts with <div id=”HTML”>, followed by a heading <h4>. A class named image <div class=”image”> is created. Within it, an image of HTML is inserted <img src=”HTML.png” alt=”HTML logo”>. A <div class=”text”> is created, containing a preformatted text tag <pre> displaying a simple HTML code. In this section, **&lt;** and **&gt;** are special characters in HTML which represent **‘<**’ and **‘>’** respectively. A line break <br/> is used to separate the contents..

# 

# Part 3:





The section above defines the structure and layout for Part 3. The code for part 3 is written inside the <body> section of the HTML document. There is no CSS code for this part.

Part 3 starts with a heading <h1> to introduce this section. Below the heading, definition list <dl> is created to hold terms and their corresponding descriptions. The list starts with a definition term <dt>, where the first term is **PLAGIARISM** and is bold using <b>. Below it, definition description <dd> is added to explain the term. Within <dd>, a line break <br> is used, followed by a hyperlink <a href="https://www.uow.edu.au/student/support-services/academic-skills/online-resources/referencing-and-citing/plagiarism/">**here**</a>for further information. After the closing of definition description </dd>, another line break <br> is added to separate the content.

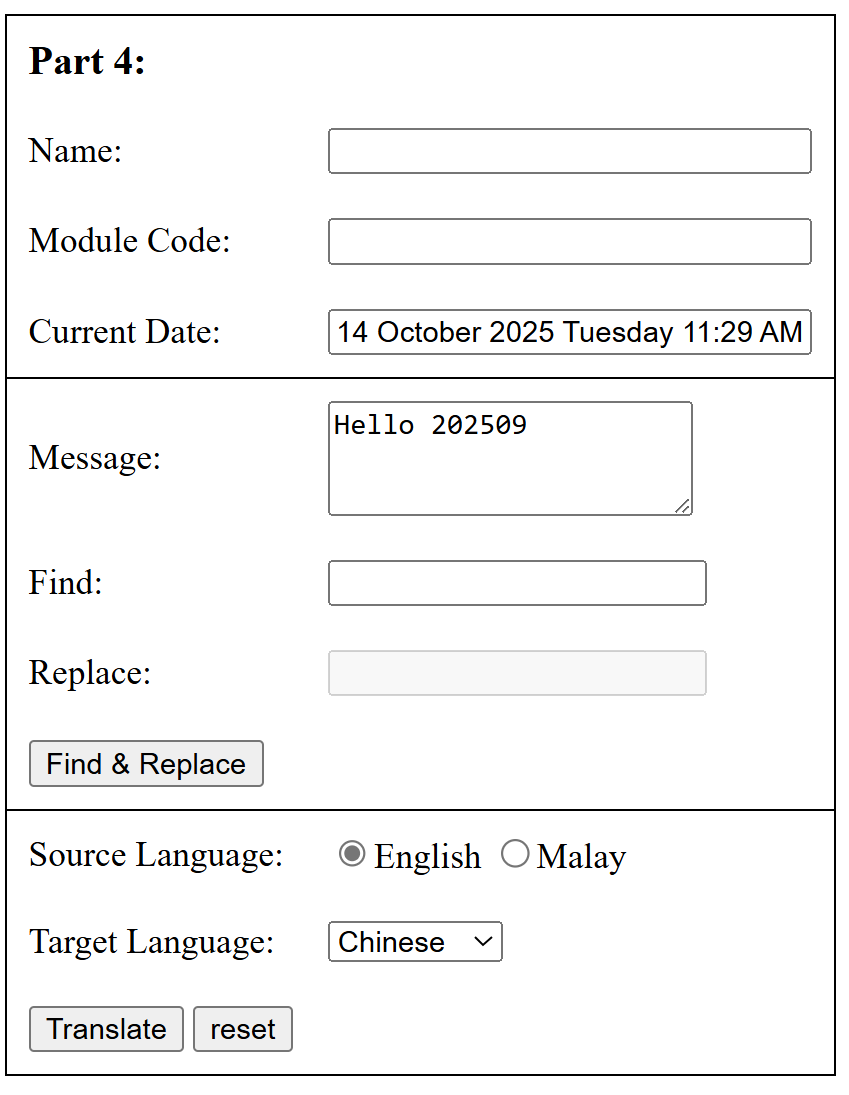
The second definition term <dt>, **ACADEMIC CONSIDERATION**, follows the same structure. The term is enclosed in <dt> and made bold using <b>, followed by definition description <dd> to explain the term. Within <dd>, after a line break <br>, a hyperlink <a href="https://www.uow.edu.au/student/admin/academic-consideration/">**here**</a> for further information. After the closing of definition description </dd>, another line break is inserted.

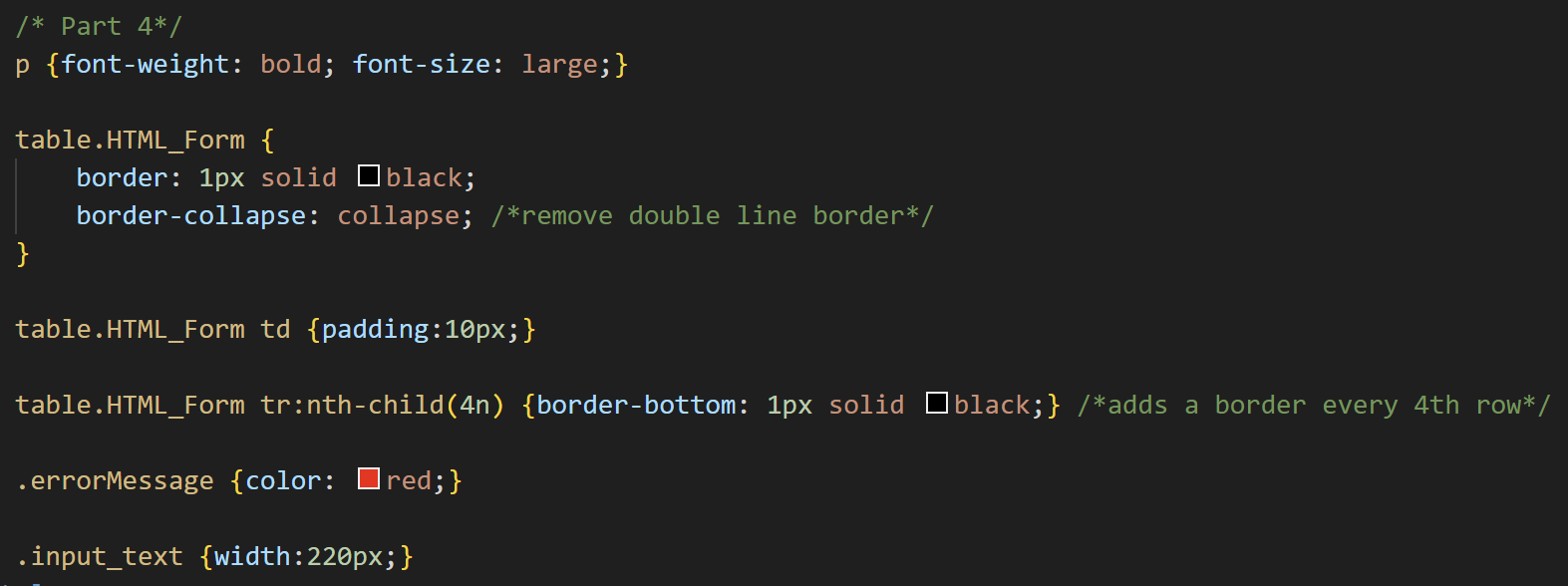
The third definition term, **STUDENT SUPPORT ADVISERS**, has two parts. The first part follows the same structure as the previous terms, a bold <b> definition term <dt>, followed by definition description <dd> that includes a line break <br> and a hyperlink <a href="https://www.uow.edu.au/student/support-services/">**here**</a> for further information. The definition description is then closed </dd>.

The second part of this term requires a table. It starts with definition description <dd> containing a table with 1 border <table border =”1”>. The table has 2 table data cells <td>, one for each adviser. Each table data <td> includes the name of the advisers which is bold using <b>, and their details such as location, phone number, availability on weekdays and email address. Each detail is segregated using a line break <br> for tidiness. The table is then closed with </table>, followed by the closing of definition description </dd> and a line break <br>.

The last term follows the same structure as the earlier terms: a bold definition term <dt>, a definition description <dd> with a line break in between, and a link for more information, followed by a closing definition description </dd> and a line break <br>

# Part 4:





The <style> section defines the CSS formatting used to style the HTML form, which is structured using a table layout.

The paragraph (p) selector is styled with bold text and a large font size to highlight the heading “Part 4:”. This helps readers recognize the start of the fourth section of the assignment, which involves creating an HTML form with various input elements.

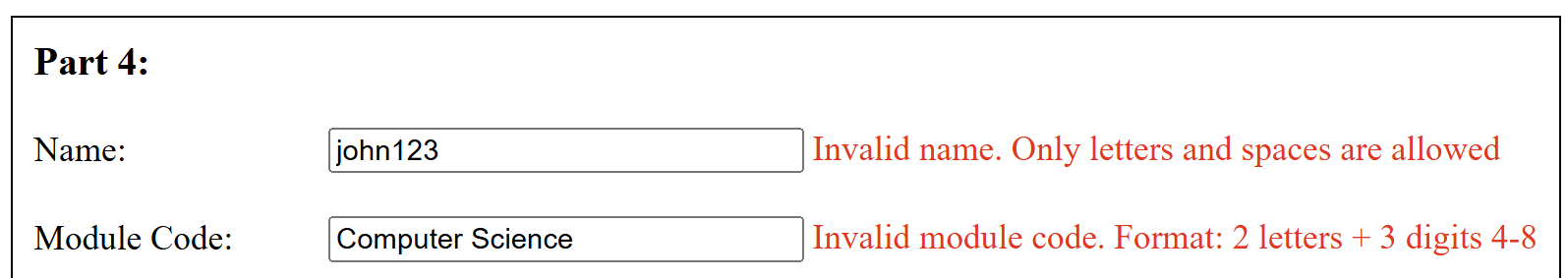
The table.HTML\_Form class targets only the form’s table, preventing interference with other tables on the page. Within this class, the border is set to 1px solid black, and border-collapse: collapse; is applied to remove the double borders between cells, resulting in a cleaner and more compact table appearance.

The td selector within the same class adds 10px of padding to each table cell, improving spacing and readability for the form’s content.

The use of table.HTML\_Form tr:nth-child(4n) draws a bottom border after every fourth row. This visually separates the form into logical sections, that being personal information (Name, Module Code, Date), text editing (Message, Find, Replace), and translation options (Source Language, Target Language, Translate/Reset buttons).

The .errorMessage class styles all error messages in red, making validation feedback noticeable and consistent throughout the form.

Finally, the .input\_text class expands certain input fields such as the Current Date field to a width of 220px, ensuring that longer text entries are displayed in full.





The <script> section handles the form validation, which ensures that the user’s input is correct before the form is submitted. The script is placed inside the <head> tag as a good coding practice which allows the functions to load before any form interaction occurs.

The validation function retrieves the user’s input values from the “Name” and “Module Code” and “comments” fields, then checks whether these inputs meet their respective conditions. If any input fails validation, an appropriate error message is displayed beside the corresponding field.

When the “Name” field is left empty (if (name\_ == null || name\_ == "")), the function triggers the error message nameError2, which displays the text “Invalid name. Field cannot be empty.” This message is shown directly on the page using the .innerHTML property allowing it to appear inline beside the field instead of an alert pop-up.

At the same time, the isValid variable is set to false, preventing the form from submitting until the error is resolved.

Once the script confirms that the “Name” field is not empty, it performs a second check to ensure that the input contains only letters and spaces. This is done using the regular expression /^[A-Za-z ]+$/, where:

* [A-Za-z] allows both uppercase and lowercase letters,
* the trailing space allows spaces between words, and
* the ^ and $ anchors ensure the entire input strictly follows this pattern.

If the name fails this check (else if (/^[A-Za-z ]+$/.test(name\_) == false)), the error message nameError1 appears beside the input, stating “Invalid name. Only letters and spaces are allowed.” The isValid variable remains false until a valid input is provided.

The subsequent validation targets the “Module Code” field, which must follow a specific format:

* The input should begin with two uppercase letters, followed by
* three digits ranging from 4 to 8 (inclusive).

Importantly, this field is optional, so it can be left blank without causing an error.

This rule is implemented with the condition:

if (modCode !== "" && /^[A-Z]{2}[4-8]{3}$/.test(modCode) == false)

* modCode !== "" ensures the validation only runs if the field isn’t empty.
* ^[A-Z]{2} enforces two uppercase letters at the start.
* [4-8]{3}$ ensures the last three characters are digits from 4 to 8.

If this condition fails, the function displays the error message modCodeError, stating “Invalid module code. Format: 2 letters + 3 digits (4–8).”

The final validation ensures that the “comments” text area is not empty, the error message “messageError” appears next to the “findReplaceBtn”, stating ”Message field cannot be empty”. This condition is the same as the validation condition for the name input field that triggers the nameError2 message, the same principle applies.

The form will only proceed with submission once all conditions for the Name, Module Code and Comments return valid inputs, ensuring the user’s entries meet the specified format.



The <form> section of the <body> represents the HTML form that users will be interacting with, where user inputs are collected in a structured table format. The table is enclosed within the <form> tag to logically group all input elements for submission.

The <form> tag includes the onsubmit="return validateForm()" attribute. This ensures that when the user clicks the submit button or presses Enter, the validateForm() function runs first, performing all input checks. If validation fails, the form submission is halted.

The table is assigned the CSS class HTML\_Form, which applies the previously defined styling: borders, row spacing, and input formatting. Each element is organized into rows (<tr>), which improves visual clarity and separates different form sections.

Header row:

The first row contains the heading “Part 4:”, spanning two columns (colspan="2") to clearly indicate the start of the form section.

Name input row:

The next row has two columns:

* Label column: <label for="Name">Name:</label> informs the user of the field’s purpose.
* Input column: <input type="text" name="Name" id="Name" class="input\_text">

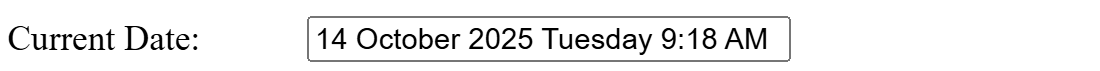
The input box itself has the following attributes:

* + type="text" defines the input as a text box (other types like radio, checkbox, submit appear later).
  + name="Name" allows server-side scripts (e.g., PHP) to identify the submitted data.
  + id="Name" connects the input to JavaScript validation and CSS styling.
  + class="input\_text" applies the CSS width of 220px for better display.

Beside the input, there are two <span> elements with IDs nameError1 and nameError2 and the class errorMessage. These spans are initially empty (innerHTML = "") and dynamically display error messages in red when validation conditions are not met, such as leaving the field blank or entering invalid characters.

Module Code row:

The following row follows the same structure: a label for the module code and a corresponding input box <input type="text" name="ModuleCode" id="ModuleCode" class="input\_text">. The associated <span> (modCodeError) dynamically displays validation errors if the input does not match the required format (two uppercase letters followed by three digits 4–8), or if the user leaves it empty, no error appears.





The function getcurrentDate() is designed to automatically display the current date and time when the webpage loads. It achieves this by using JavaScript’s built-in Date object, which provides methods to retrieve and format time-related information.

The function begins by creating a new Date instance with var today = new Date(); This object holds the current date and time based on the user’s system clock.

From this object, several components are extracted:

* getDate() retrieves the day of the month (1–31).
* getFullYear() returns the four-digit year.
* getHours() retrieves the current hour (0–23, using 24-hour format).
* getMinutes() retrieves the current minute (0–59).

To convert the numerical digits to an accessible format for users to read, the script defines two arrays, one for month names and another for weekdays. They are represented by the var monthList and weekdayList respectively.

The methods getMonth() and getDay() return the current month (0–11) and weekday (0–6) as indices, which are then used to access the corresponding names from these arrays.

Next, the function ensures the time is displayed in 12-hour format with proper leading zeros and an AM/PM indicator:

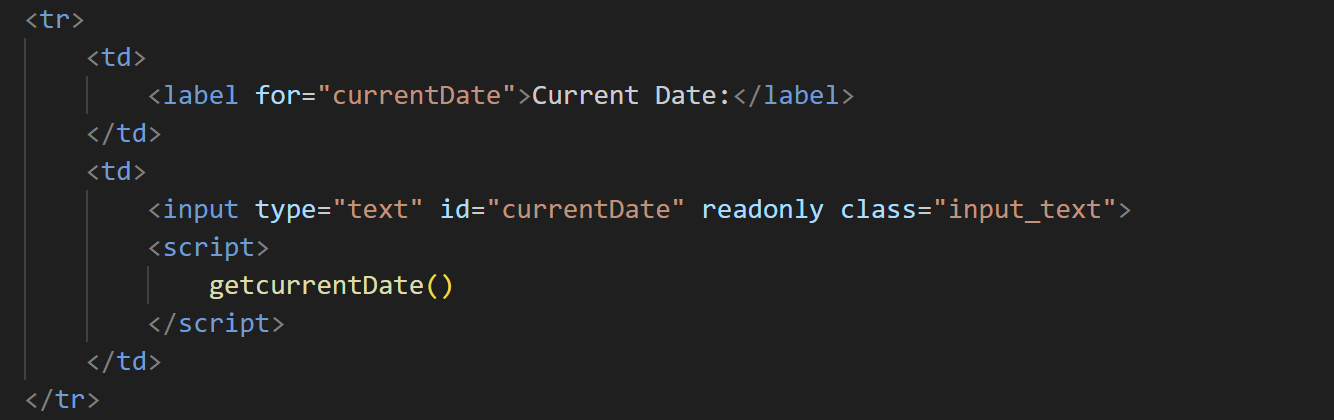
* If the minutes are less than 10, a "0" is added in front (e.g., 09:05 instead of 9:5).
* The expression hours >= 12 ? 'PM' : 'AM' determines whether it’s morning or afternoon.
* The conversion hours = hours % 12 || 12; changes 24-hour time to 12-hour format while ensuring midnight (0) displays as 12.

The final date-time string is formatted based on the assignment’s criteria through the following concatenation:

var formattedDate = day + " " + monthName + " " + year + " " + weekday + " " + hours + ":" + formatted\_minutes + " " + ampm;

This creates an output like: 1 October 2025 Wednesday 1:47 PM

Finally, the line “document.getElementById("currentDate").defaultValue = formattedDate;” assigns this formatted string as the default value for the input field with the ID "currentDate". Using defaultValue instead of value ensures that even if the user resets the form, the date and time remain visible.



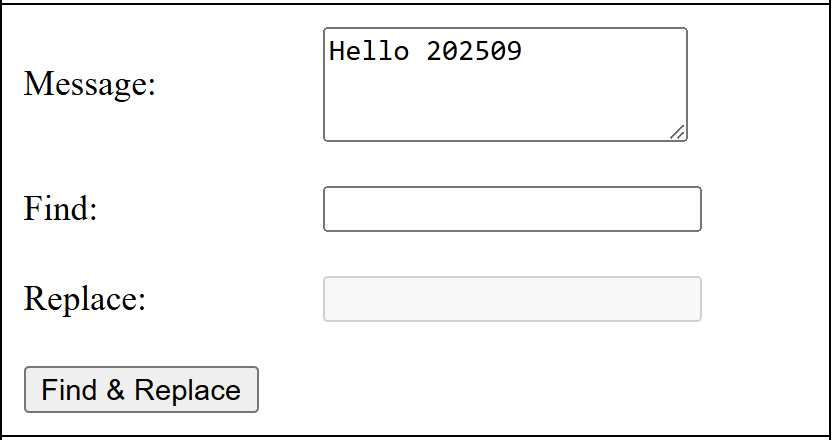
The above section of code is part of the <body> where the <form> and <table> elements define the structure of the current date input field. This section continues from the previous input rows for the user’s name and module code.

A new table row <tr> is created to display the Current Date label and its corresponding input box. The first table data cell <td> contains a <label> element with the for="currentDate" attribute, linking it to the input field for accessibility and readability purposes.

The second <td> holds the actual <input> element with the type text, and its id set to currentDate. This id acts as a reference point for the JavaScript function getcurrentDate(), which dynamically assigns the current date and time to this input field. The readonly attribute ensures users cannot manually alter the field’s content, keeping the displayed value accurate and system-generated. The class="input\_text" is included for consistent styling across form fields.

Immediately below the input field, the <script> tag calls the getcurrentDate() function. Placing this function call right after the input ensures that the element is already loaded in the DOM before the script executes. This prevents any potential runtime errors or null references.

Finally, the table row is properly closed with </td> and </tr> tags, signifying the end of this section of the table.





The above section of code defines the findAndReplace() function, which performs a dynamic find-and-replace operation on the text entered in the “Comments” text area. It searches for the string entered in the “Find” input box and replaces all matching instances with the text entered in the “Replace” box.

At the start of the function, several variables are declared using document.getElementById() to reference the following HTML elements:

* findInput refers to the “Find” text box, where the user specifies the target string.
* replaceInput refers to the “Replace” box, where the user enters the replacement text.
* textArea refers to the “Comments” input field that contains the message to be edited.
* replaceResult is the element used to display status messages, such as the number of replacements made or when no matches are found.
* findReplaceBtn represents the button that triggers the find-and-replace operation.

As per the assignment requirement, the “Replace” input field must remain disabled until the user types something in the “Find” field. To achieve this, an event listener is attached to findInput that monitors the "input" event. When the value in findInput is blank (determined using .trim() to ignore whitespace), the replaceInput field is disabled and cleared, and the replaceResult message is reset. This ensures that users cannot attempt a replacement with an empty “Find” field.

The click event listener on findReplaceBtn handles the main logic. When triggered, it first retrieves the trimmed values from findInput and replaceInput, as well as the text content of the “Comments” area. If the “Find” box is empty, the function exits immediately using return, thus preventing further execution.

If the message contains the specified find text (checked using message.includes(findText)), the code performs a global replacement using the replaceAll() method. This creates a new string, newMessage, with every occurrence of the target text replaced by the new one.

To provide user feedback, a matchCount variable calculates how many replacements were made by splitting the original string based on the find text and subtracting one from the resulting array length. This number is then displayed in the replaceResult element, informing the user of how many replacements occurred.

If no matches are found, the program updates the replaceResult message to “No matches found,” ensuring clear communication of the outcome.



The above section of code defines the HTML structure for the Find & Replace form, which allows the user to search for a specific word or phrase in the “Message” text area and replace it with another input value. The form elements are neatly organized in each table row <tr> tag for readability and consistent layout. The first row contains a label and a multi-line text input field (<textarea>), which serves as the main editable area where users can view or modify their text.

* The <label> tag, linked using the for attribute, improves accessibility by associating the label with its respective input element.
* The <textarea> element uses the attributes rows=”3” and cols=”20” to specify its visible dimensions as specified in the assignment. It also has a default value (“Hello 202509”) to give users a starting point for testing the find-and-replace functionality.
* The id="comments" identifier connects this text area to the JavaScript function that manipulates its content.

The second row defines a text input where users specify the term they want to locate in the message.

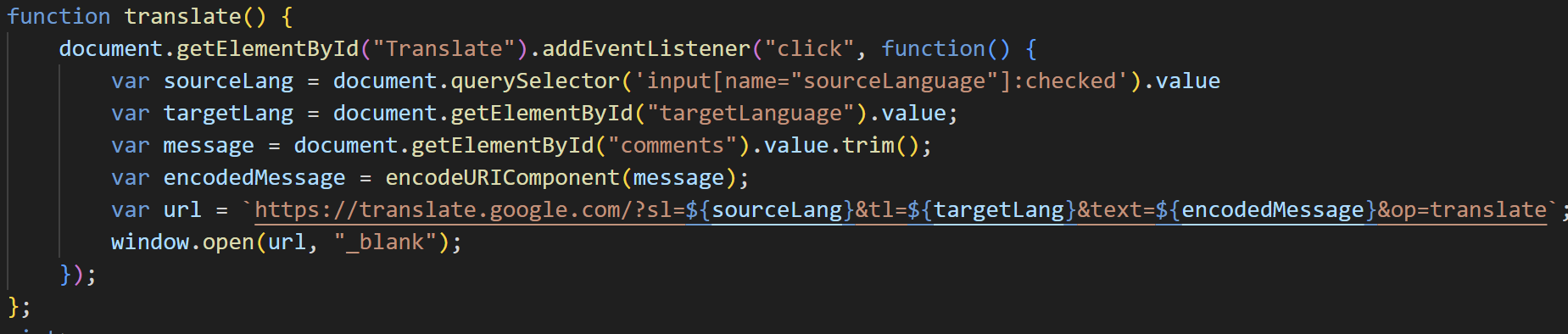
* The name attribute provides a key for backend reference (useful if the form were submitted to a server).
* The id attribute allows JavaScript to directly reference this element.
* The for="find" association in the label maintains accessibility standards, ensuring screen readers can correctly link label and input.

The third row defines the “Replace” input field, which will hold the text that replaces the term found in the message.

* Initially, the disabled attribute prevents user interaction until the “Find” box contains text.
* This dynamic behavior is controlled by the event listener within the findAndReplace() JavaScript function.
* This approach ensures users cannot attempt to replace text before specifying a target, improving both usability and data integrity.

The final row brings together user interaction and output feedback:

* The <button> element triggers the find-and-replace process when clicked. It is linked to the function through the JavaScript event listener set within findAndReplace().
* The <span> element displays status messages (e.g., “2 replacements made” or “No matches found”), using the errorMessage class for the text to appear red.
* The embedded <script> tag calls the findAndReplace() function immediately after the button is defined, ensuring the DOM elements exist before the function attaches its event listeners. This placement prevents JavaScript errors that can occur if a script runs before the related HTML elements are fully loaded.



The above section of code defines the function translate() which is responsible for translating the text content entered in the message text area into another selected language. The translation is performed by redirecting the user to Google Translate in a new browser tab, with the relevant source language, target language, and message text embedded into the translation URL.

Within the function, we begin by adding an event listener to the “Translate” button through document.getElementById("Translate").addEventListener("click", function(){...}); This ensures that the translation process only executes when the user clicks the button.

We first establish several variables to capture the necessary input values. The variable sourceLang retrieves the value of the selected source language radio button using document.querySelector('input[name="sourceLanguage"]:checked').value. The targetLang variable retrieves the value from the target language dropdown list via document.getElementById("targetLanguage").value. The variable message stores the content from the “comments” text area using its ID and applies .trim() to remove any leading or trailing spaces.

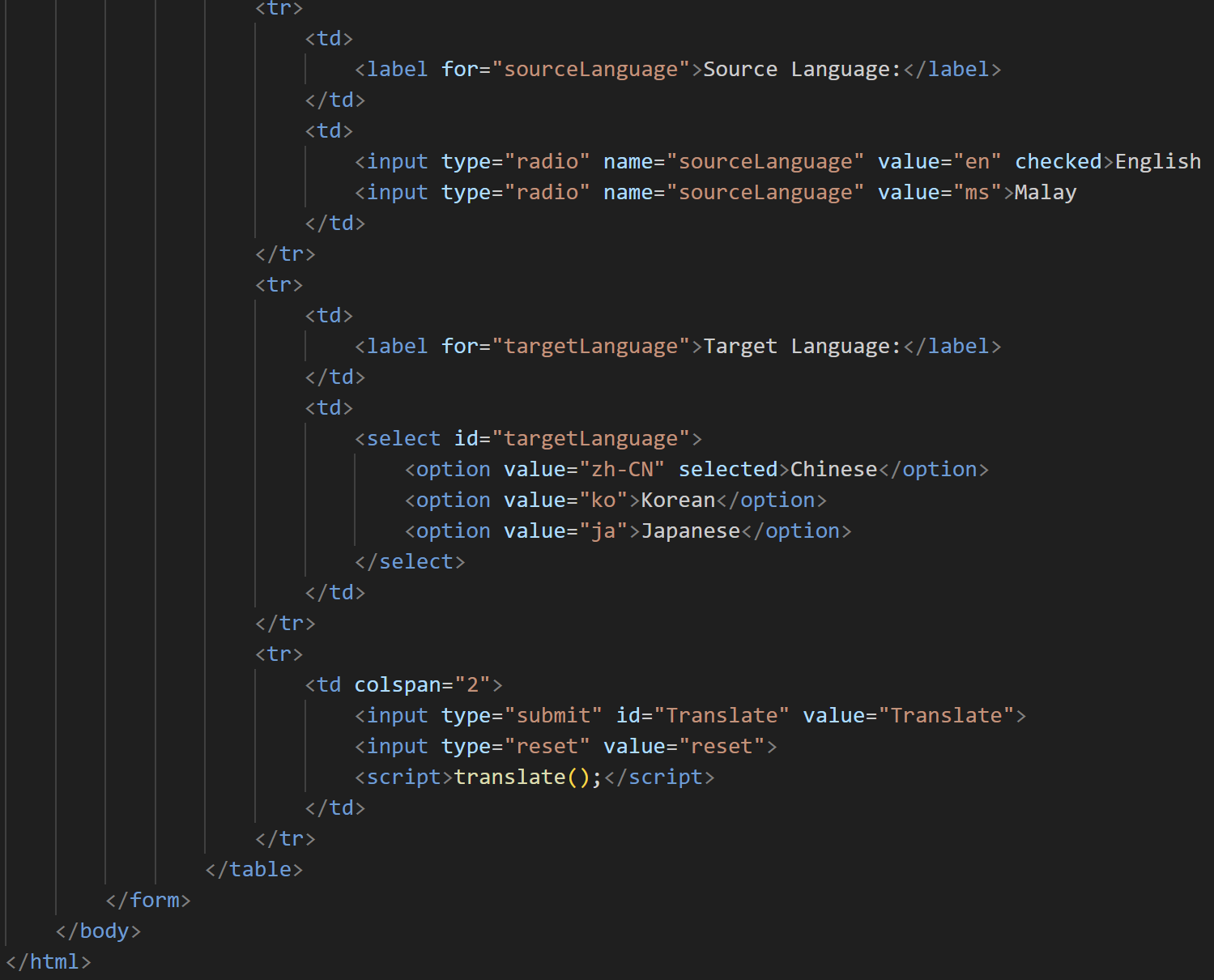
To ensure that the message content can be safely included within a URL, it is encoded using JavaScript’s built-in function encodeURIComponent(message). This step converts any special characters, spaces, or symbols into a format that can be correctly interpreted by web browsers. The result is stored in the variable encodedMessage.

Next, we construct the URL for Google Translate using a template literal:

var url = `https://translate.google.com/?sl=\${sourceLang}&tl=\${targetLang}&text=\${encodedMessage}&op=translate\`

The variable sl, tl, and text represent the source language, target language, and encoded message respectively. By embedding the variables directly into the URL, the script dynamically adjusts the translation settings according to the user’s selections and input.

Finally, the statement window.open(url, "\_blank"); instructs the browser to open a new tab containing the constructed Google Translate URL. This allows the user to view the translation result without leaving or refreshing the current page.



The above section represents the final portion of the form, handling the translation options and submission buttons.

The first table row defines the Source Language input, allowing users to select between English and Malay using radio buttons. Both share the same name attribute, ensuring only one can be selected at a time. English is pre-selected by default through the checked attribute.

The next table row provides a Target Language dropdown list using the <select> tag, containing language options such as Chinese, Korean, and Japanese. Chinese is set as the default through the selected attribute. The dropdown list’s id="targetLanguage" is used by the JavaScript function to retrieve the selected language code when performing the translation.

The final table row contains two form buttons: the Translate button (<input type="submit">) and a Reset button (<input type="reset">). The Translate button triggers the translate() JavaScript function that opens a new browser tab with Google Translate, passing in the user’s selected source and target languages along with their message. The Reset button clears all current inputs, restoring the form to its initial state. The <script>translate();</script> line initializes the function when the page loads, ensuring that the event listener is properly attached to the Translate button.

The section concludes with the closing tags for the table, form, and document body, marking the end of the webpage structure.