The Universal Acceptance Curriculum: The Micro-Learning Module on Unicode Programming Fundamentals.

This is the 1st Edition.

1. UA Micro-Learning Module 1: Unicode Programming Fundamentals.

Welcome to this micro-learning module dedicated to the essential concepts and techniques of Unicode programming.

In the field of modern software development, knowledge of Unicode is essential as it serves as the foundation for effectively representing and manipulating characters from diverse languages, scripts, and symbol systems. Mastery of Unicode is not only critical for building robust applications but is also pivotal in the pursuit of Internationalization which extends to Universal Acceptance, where software systems can embrace and accommodate the richness of linguistic diversity.

In this micro-learning module, we will discuss the fundamental principles of Unicode, covering key areas such as encoding, decoding, character manipulation, and the effective handling of various Unicode transformation formats. By the conclusion of this micro-learning module, you will have acquired a solid foundation in Unicode programming. With this newfound knowledge, you will gain the confidence to write Java and Python codes that effortlessly meets the demands of Unicode requirements, allowing for seamless handling of Unicode in your programming tasks.

2. Objectives:

Upon completion of this micro-learning UA module, students will be able to:

- Understand the concept of Unicode;
- Comprehend Unicode encoding and its advantages;
- Acquire a solid grasp of Unicode code points and code units;
- Understand the concept of surrogate pairs and their usage;
- Develop a comprehensive understanding of key Unicode terminologies;
- Work with Unicode in programming:
  - How to represent Unicode characters in a character data type
  - How to create, input, and concatenate Unicode strings
  - How to display Unicode strings.
- Appreciate the significance of Unicode in promoting linguistic diversity and cultural exchange.

3. Targeted Audience

This micro-learning module is intended for undergraduate students enrolled in IT, Computer Science or related programs.

4. Prerequisites

This micro-learning module has no prerequisite and is recommended for integration into the introductory or fundamental programming course within existing Information Technology, Computer Science and related programs curricula of higher education institutions.

5. Micro-Learning Materials

Alongside this micro-learning material and its video recorded presentation, students are encouraged to utilize resources listed in the reference section, as well as other relevant sources. These materials including examples code and assignments are available from the url(link to the micro-learning module materials):

6. Micro-Learning Module Policy

This micro-learning module policy adopts the policy of the course into which it is recommended for integration.

7. Micor-Learning Module Assessment

The instructor has the flexibility to employ formative assessments, or summative assessments or a combination of both, determining the weight of each assessment, in accordance with the course policy to which the micro-learning module is integrated.

8. Course Timing

The table provided below outlines the timing for module topics within the UA micro-learning module. It is important to highlight that the cumulative classroom duration for all module topics in this UA micro-learning module amounts to **1 hour** and **27 minutes**. Part 1: Unicode Basics requires **39 minutes**, while Part 2: UA Path - Unicode Programming Fundamentals in Java requires **48 minutes**, and Part 3: UA Path - Unicode Programming Fundamentals in Python also requires the same duration.

Cells in the column that are not necessary for a particular topic are indicated with a hyphen (-) .

Module Topics Title	Lecture (Minutes)	Activity /Lab (Minutes)	Knowledge Check (Minutes)	Total Classroom Time (Minutes)
Overview of Encoding Schemes.	15	-	5	20
Why do we need Unicode?	4	-	-	4
Basic Unicode Terminologies and Understanding Unicode Encoding Schemes.	6	-	3	9
What is Surrogate Pair?	3	-	2	5
Character data type for Unicode code points.	1	-	-	1
Unicode Programming Fundamentals in Java				
char data type in Java.	2	1	-	3
Working with surrogate pairs in Java.	4	2	-	6
How to store and process Unicode characters or strings in Java.	5	10	-	15
Iterating over code points.	2	3	-	5
Concatenating Unicode Strings.	2	2	-	4
Introduction to Unicode Charts.	5	-	-	5
Working with Unicode Formats : Storing Unicode Data in UTF-8 Format Files.	5	5	-	10

Unicode Programming Fundamentals in Python				
Character as str data type in Python.	2	1	-	3
Working with surrogate pairs in Python.	4	2	-	6
How to store and process Unicode Characters or Strings in Python.	5	10	-	15
Iterating over code points.	2	3	-	5
Concatenating Unicode Strings.	2	2	-	4
Introduction to Unicode Charts.	5	-	-	5
Working with Unicode Formats : Storing Unicode Data in UTF-8 Format Files.	5	5	-	10

9. Mode of Integrating Micro-learning Materials into the Curriculum

Educators or trainers can choose the timing for delivering micro-learning module topics, either by integrating them alongside relevant course topics in the existing Computer Science and Information Technology curriculum or by presenting them at the conclusion of a course unit.

10. Overview of Encoding Schemes

Character encoding schemes play a fundamental role in representing characters within digital systems, making a solid understanding of them is vital for working with text. In digital systems, three prominent character encoding schemes are commonly employed: ASCII, ISO 8859, and Unicode. Each encoding scheme possesses distinct characteristics and serves specific purposes, catering to the diverse needs of character representation in digital contexts.

ASCII is a widely used character encoding scheme that represents text in digital systems. It uses 7 bits to encode characters, allowing for a total of 128 characters. Originally designed for the English language, ASCII includes control characters and printable characters commonly used in English text, such as letters, numbers, punctuation marks, and special symbols. ASCII enjoys wide compatibility across systems and programming languages, serving as a foundational encoding scheme. However, it has limitations in supporting non-English languages and lacks the comprehensive character set found in more modern encoding schemes like Unicode. ASCII is a widely adopted scheme that represents English text using 7 bits and includes 128 characters. It is compatible with many systems but lacks support for non-English languages. ISO 8859 extends ASCII by adding variations that cover specific regions and languages, providing limited multilingual support. However, Unicode surpasses both ASCII and ISO 8859 by offering a comprehensive encoding standard for characters from all writing systems worldwide. Unicode ensures compatibility across languages, scripts, and symbols, enabling multilingual communication and global software development. It has become the preferred choice for modern applications, supporting diverse linguistic and cultural requirements.

ISO 8859 is a set of character encoding schemes that extend the ASCII encoding scheme to support additional languages. Each ISO 8859 variation, such as ISO 8859-1, ISO 8859-2, etc., covers specific regions or languages. These variations introduce additional characters beyond the ASCII set, allowing for better language support. ISO 8859 is generally backward compatible with ASCII, preserving the ASCII character set in the lower 7 bits while incorporating additional characters in the higher bits. However, ISO 8859 has limitations in terms of comprehensive multilingual coverage and is not as widely used as Unicode, which provides a more comprehensive and globally compatible character encoding standard. ISO 8859 extends the range of characters but still has limitations in representing non-English languages effectively.

Unicode is a comprehensive character encoding scheme that represents characters from a vast array of writing systems worldwide. It provides a unified standard for character representation across languages, scripts, and symbols. With a vast range of characters assigned unique code points, Unicode ensures compatibility and facilitates multilingual communication. It supports diverse languages, scripts, emojis, and more, accommodating the linguistic and cultural diversity of global content. Unicode has become the preferred encoding scheme in modern software development, enabling universal character handling, globalization, and localization efforts. Unicode is a character encoding standard that assigns a unique code point to every character from various writing systems, including alphabets, symbols, and emojis. The Unicode standard currently supports over 143,000 characters, which are represented by code points ranging from U+0000 to U+10FFFF.

While they all serve the purpose of character representation in digital systems, there are significant differences among them. The following is a summary highlighting the comparative differences among the three encoding schemes we discussed, focusing on their respective range of character set support, language support, compatibility, scope, and purposes.

Range of character set support:

- ASCII uses a 7-bit character set, providing codes for 128 characters. It primarily focuses on the English language and includes control characters and printable characters commonly used in English text.
- ISO 8859: ISO 8859 extends the character set beyond ASCII, providing several variations (such as ISO 8859-1, ISO 8859-2, etc.) to support additional languages. Each variation covers a specific region or language, expanding the range of characters available. The ISO 8859 standard defines a series of character sets for various languages and regions. However, ISO 8859 specifies a total of 15 character sets, numbered as ISO 8859-1 to ISO 8859-15. Each ISO 8859 character set within the ISO 8859 standard can represent 256 different characters.
- Unicode is a comprehensive encoding scheme designed to represent characters from a vast array of writing systems worldwide. It includes a vast range of characters, symbols, scripts, and emojis, with each character assigned a unique code point. The

Unicode standard currently supports over 143,000 characters, which are represented by code points ranging from U+0000 to U+10FFFF.

Language Support:

- ASCII is primarily designed for the English language. It does not have dedicated support for characters from non-English languages or special characters commonly used in other languages.
- ISO 8859 variations provide limited support for non-English languages by incorporating additional characters specific to different regions. However, the language coverage varies across the ISO 8859 variations, and it may not fully cater to all languages and writing systems.
- Unicode: Unicode provides extensive multilingual support, accommodating characters from various languages and scripts globally. It aims to ensure comprehensive representation and communication across different languages.

Compatibility:

- ASCII is widely compatible across different systems and programming languages. It serves as a foundational encoding scheme and is supported by almost all modern computing environments.
- ISO 8859 variations are generally backward compatible with ASCII. They preserve the ASCII character set in the lower 7 bits while introducing additional characters in the higher bits. This compatibility allows systems and software designed for ASCII to handle ISO 8859-encoded text without significant issues.
- Unicode: Unicode is designed to be compatible with ASCII, ensuring a smooth transition and backward compatibility. It allows systems and software designed for ASCII to handle Unicode-encoded text without significant issues.

Scope and Purpose:

- ASCII: ASCII is a basic encoding scheme suitable for English text and simple digital communication.
- ISO 8859 variations expand the character set to support additional languages but still have limitations in terms of comprehensive multilingual coverage.
- Unicode is the most comprehensive encoding scheme, addressing the limitations of ASCII and ISO 8859. It provides a unified standard for character representation, enabling global compatibility and multilingual communication.

The principles of representing plain text and the Unicode character-glyph model are covered in Module 2 (Unicode Advanced Programming).

11. Why do we need Unicode?

Unicode has gained significant popularity and is increasingly used as the standard for character encoding in modern software development. It addresses the limitations of ASCII and ISO 8859, offering comprehensive multilingual support and enabling globalization and localization efforts in software systems. While ASCII focuses on the English language, ISO 8859 extends the character set to support additional languages. Unicode surpasses both

ASCII and ISO 8859, providing a unified standard for character representation across all languages and scripts globally.

12. Basic Unicode Terminologies and Understanding Unicode Encoding Schemes

Unicode provides various encoding schemes to represent characters from different writing systems and languages. Below is a summary of the prevalent Unicode encoding schemes:

12.1. Basic Unicode Terminologies:

Unicode Scalar Value: refers to the numerical representation of individual characters within the Unicode character set. It represents a unique and standardized code point assigned to each character. These values are represented in hexadecimal notation and range from 0 to 10FFFF (hexadecimal).

Code Unit: refers to the basic unit of storage or representation for encoded characters. It is the smallest individually addressable element within a given encoding scheme.

Code Unit Sequence: refers to a consecutive series or sequence of code units within a character encoding. It represents a sequence of individual units that collectively encode characters or textual data.

12.2. Understanding Unicode Encoding Schemes

12.2.1. UTF-8:

- UTF-8 is a widely used variable-length encoding scheme.
- It uses one to four bytes to represent characters, with ASCII characters represented by a single byte.
- UTF-8 is backward compatible with ASCII and offers efficient storage and transmission of commonly used characters.
- In UTF-8, a code unit is typically 8 bits (1 byte) and represents a single Unicode character. However, it can be larger for characters outside the Basic Multilingual Plane (BMP).

12.2.2. UTF-16:

- UTF-16 is a fixed-length encoding scheme that uses 16-bit code units.
- It can handle the entire Unicode character set, including characters outside the Basic Multilingual Plane (BMP).
- UTF-16 is commonly used in programming languages and platforms that internally represent characters as 16-bit code units.
- In UTF-16, a code unit is 16 bits (2 bytes) and can represent either a single Unicode character within the BMP or a surrogate pair that encodes characters beyond the BMP.

12.2.3. UTF-32:

- UTF-32 is a fixed-length encoding scheme where each character is represented by a four-byte code unit. UTF-32 has big-endian and little-endian variants based on the order of bytes within the code unit, analogous to UTF-16. These variants are known as UTF-32BE and UTF-32LE respectively.
- It provides a direct one-to-one mapping between code points and code units.
- In UTF-32, a code unit is 32 bits (4 bytes) and directly corresponds to a single Unicode character, regardless of its code point value.

Choosing the appropriate Unicode encoding scheme depends on factors such as storage efficiency, compatibility requirements, and the specific context or platform being used. Understanding the characteristics and differences between these encoding schemes is crucial for accurately representing and manipulating characters from diverse writing systems in a consistent manner.
UTF-8 is a widely used character encoding scheme that allows the representation of all Unicode code points. It offers efficient storage and transmission of Unicode characters by utilizing variable-length encoding. Here's a summary of the key points:

- UTF-8 uses variable-length encoding, where different Unicode characters require different numbers of bytes for representation.
- ASCII characters are represented using a single byte, maintaining compatibility with the ASCII character set.
- Characters outside the ASCII range use multiple bytes, allowing for the representation of diverse characters from different writing systems.
- Each Unicode code point is represented by a sequence of bytes, with the number of bytes determined by the code point value.
- UTF-8 follows specific encoding rules to ensure self-synchronization and proper decoding of byte sequences.
- ASCII characters (0 to 127) are encoded using a single byte, while higher code points require multiple bytes.
- UTF-8 is backward compatible with ASCII and provides efficient storage and transmission of commonly used characters.
- It is widely supported by modern systems, programming languages, and web standards.

Understanding UTF-8 encoding is crucial for effectively working with Unicode data in programming languages. It enables accurate representation and manipulation of characters from various writing systems, ensuring compatibility and seamless interaction with different systems and applications. By grasping the nuances of UTF-8 encoding, developers can confidently handle Unicode data, ensuring accurate character representation and consistent behavior across different platforms and programming environments.

13. What is Surrogate Pair?

A surrogate pair is a UTF-16 encoding mechanism used to represent supplementary characters outside the Basic Multilingual Plane (BMP). It involves using a combination of two 16-bit values, known as high surrogates and low surrogates, to represent characters in the supplementary planes. Surrogate pairs are important for accurately encoding and handling characters from less commonly used scripts and specialized symbols.

Example 1: Examples of surrogate pairs used to represent emojis.

- Grinning Face with Smiling Eyes Emoji:
  - Emoji: 😄

- Unicode Code Point: U+1F604
- Surrogate Pair: U+D83D U+DE04

- Thumbs Up Emoji:
  - Emoji: 👍
  - Unicode Code Point: U+1F44D
  - Surrogate Pair: U+D83D U+DC4D

14. Character Data Type for Unicode Code Points

The character data type commonly used for representing Unicode code points is char. In many programming languages, including Java, C++, C#, and Python, the char data type is used to store a single Unicode character.

Note:

- The char data type typically uses a fixed number of bits to represent the Unicode code unit, allowing for the storage of a wide range of characters.
- Unicode code points can be assigned to a ***char*** variable, allowing for character manipulation, comparison, and other applicable operations.
- It is important to note that the specific implementation and size of the ***char*** data type may vary across programming languages and platforms.
- Throughout this module, we delve into the basics and nuances of utilizing the ***char*** data type for handling Unicode code points specifically within the Java and Python programming environments. By focusing on the two programming languages, the module aims to equip learners with a solid understanding of how to effectively work with Unicode characters or strings using the ***char*** data type, enabling you to develop robust and script or language-aware software solutions.

Unicode Programming Fundamentals in Java

1. Char Data Type in Java

In Java, the char data type is used to represent a single code unit. Java supports Unicode by using the UTF-16 encoding scheme, where each Unicode code point is represented by one or two char values. It is designed to store Unicode characters and uses 16 bits (2 bytes) of memory. This 16-bit size allows the ***char*** data type to cover a wide range of Unicode code points.

The following Java code demonstrates how char data type can be used in Java to store Unicode characters.

Example 2: Creating/Storing Unicode Characters in Java:

```
public class UnicodeExample
{
   public static void main(String[] args)
    {
      char unicodeChar1 = '\u0627'; // Unicode character for Arabic letter "alef" (ا)
       char unicodeChar2 = '\u1200'; // Unicode character for Ethiopic letter "ሀ" (ha)
       char unicodeChar3 = '\u0411'; // Unicode character for  Cyrillic letter 'Б' (be)
      // Printing the stored Unicode characters
      System.out.println("Arabic Letter:" + unicodeChar1); // Output: ا
      System.out.println("Ethiopic/Ge'ez Letter:" + unicodeChar2); // Output: ሀ
      System.out.println("Cyrrilic Letter: " +  unicodeChar3); // Output: Б
    }
}
```

2. Working with Surrogate Pairs in Java

Java provides built-in methods to encode and decode surrogate pairs. Surrogate pairs are used to represent Unicode characters that fall outside the Basic Multilingual Plane (BMP), which includes characters up to U+FFFF., Emoji characters often fall outside the BMP and require surrogate pairs[1].

Below is a sample code snippet that demonstrates storing and printing the Unicode character using surrogate pairs in Java. It showcases the usage of surrogate pairs to handle Unicode characters beyond the Basic Multilingual Plane, i.e. it uses a combination of two ***char*** values to represent emoji (😀).

When storing surrogate pairs in Java using the ***char*** data type with escape sequences, consider the following main points:

- **Surrogate Pair Escape Sequence:** you can use Unicode escape sequences to represent surrogate pairs. A surrogate pair is represented by two Unicode escape sequences, one for the high surrogate and one for the low surrogate.
- **High Surrogate Escape Sequence:** The Unicode escape sequence for the high surrogate is \u####, where #### represents the hexadecimal value of the high surrogate. The high surrogate range is from \uD800 to \uDBFF.

[1] https://stackoverflow.com/

- **Low Surrogate Escape Sequence:** The Unicode escape sequence for the low surrogate is also \u####, where #### represents the hexadecimal value of the low surrogate. The low surrogate range is from \uDC00 to \uDFFF.
- **Combining Escape Sequences:** To store a surrogate pair, you need to combine the escape sequences for the high and low surrogates together.

Example 3: Separately storing the surrogate pairs:

char lowSurr = '\uD83D'

charhighSurr = '\uDE00'

String Formation from surrogate pairs:

You can form a String object containing surrogate pairs using escape sequences. By concatenating the escape sequences for the high and low surrogates together the surrogate pair can be formed. Below is a Java code to illustrate how Java's String class can be used to store surrogate pairs.

Example 4: Using surrogate pairs in Java.

```
public class SurrogatePair
{
   public static void main(String[] args)
   {
        String unicodeCharEmoji = "\uD83D\uDE00"; // Unicode character for 😀 (represented by surrogate pairs)
       // Printing the emoji character
       System.out.println(unicodeCharEmoji); // Output: 😊


   }
}
```

3. How to Store and Process Unicode Characters or Strings Java:

Unicode string processing in Java involves handling strings that contain characters from various Unicode code points, including those outside the Basic Multilingual Plane (BMP). Java provides built-in support for Unicode string processing through its String class and related utility classes.

The following are commonly used Unicode string processing in Java:

- String creation
- String length
- Code point count
- Accessing individual code points
- Iterating over code points

3.1. Unicode string creation

Java's String class supports the creation of strings containing Unicode characters. Strings can be created using Unicode escape sequences, surrogate pairs, or directly using the characters themselves.

Note:

- Java's String class uses UTF-16 encoding to represent Unicode characters.
- UTF-16 is a variable-length encoding that uses one or two 16-bit char values to represent characters.
- You can store Unicode characters in string literals by directly including them as characters or using escape sequences as shown in the third bullet below.

Example 5: Examples on Creating Unicode Strings:

- **Using Unicode escape sequence :**

  String unicodeString = "\U178x"; //Khmer ឨ

- **Surrogate pair for "smiling face with smiling eyes" emoji (😊) :**

  String emojiString = "\uD83D\uDE0A"

- **Directly using Unicode characters or string literals:**

  If you cannot switch your keyboard to the script used in the example string, you can use a language translator tool like Google Translate to obtain the string literals for the code snippet.

  String plainString1 = "Hello, 世界!";

  String plainString2 = "Hello, ዓለም!";

  String plainString3 = "Hello, عالم!"

  The accompanying Unicode string for "Hello" represents "World" in Chinese, Ethiopic(Amharic), and Arabic respectively.

3.2. String length

The length() method of the String class returns the number of Unicode code units (16-bit char values) in the string. However, note that this may not be the same as the number of Unicode characters or code points.

Example 6: To Find the Number of Code Units in a Unicode String:

For plain strings:

String str = "Hello, عالم!";

int codePointCount = str.codePointCount(0, str.length());

System.out.println(codePointCount);

// Output: The string length is: 12

For non-plain strings such as surrogate pairs:

```
String emojiString = "\uD83D\uDE0A";
int length = emojiString.length();
System.out.println(" The emoji string length is: " + length);
 //Output: The emoji string length is: 2
```

3.3. Code Point Count

To obtain the number of Unicode code points in a string, you can use the **codePointCount()** method of the String class. This method considers surrogate pairs as a single code point.

Example 7: To Find the Number of Code Points in a Unicode String.

Code point count for plan strings:

```
String str = "Hello, 世界!";
int codePointCount = str.codePointCount(0, str.length());
System.out.println("Code point count is : " + codePointCount );
 //Output: Code point count is : 10
```

Code point count for non-plain strings such as surrogate pairs:

```
String emojiString = "\uD83D\uDE0A";
int  codePointCountEmoji= emojiString.codePointCount(0, emojiString.length());
System.out.println("Code points count is: " + codePointCountEmoji);
 //Output: Code points count is: 1
```

3.4. Accessing Individual Code Point

The charAt() method of the String class returns the 16-bit char value at the specified index. However, to access individual Unicode code points, including those represented by surrogate pairs, you can use the **codePointAt()** method. This method returns the code point at the given index, taking into account surrogate pairs.

Example 8: To find the code point at a given index:

```
String str = "Hello, 世界!";
int codePoint = str.codePointAt(7);
System.out.println("Code point at Index 7 is:" + codePoint);
//Output: Code point  at index 7 is 19990  i.e. Unicode code point of '世' in decimal.
```

4. Iterating Over Code Points

The codePointAt method in Java is used to retrieve the Unicode code point value of a character at a specified index within a string. It is particularly useful when dealing with Unicode characters that are represented by multiple code units. This method calculates and returns the index of the code point that is located a specified number of code points away from a given index. In other words, it enables you to navigate through a string by code points rather than by individual characters. This

can be particularly useful when working with non-Latin scripts or when dealing with characters that require multiple code points to represent or surrogate pairs.

Example 9:Iterating Individual Code Points within a Unicode String.

```
public class CodePointIteration
{
   public static void main(String[] args) {
      String text = "Hello, عالم!";
      // Iterate over code points
      for (int i = 0; i < text.length(); ) {
         int codePoint = text.codePointAt(i);
         System.out.println("Code point: " + codePoint);
         // Increment the index based on the code point
         i += Character.charCount(codePoint);
      }
   }
}
```

5. Concatenating Unicode Strings

In Java, similar to string concatenation in ASCII, Unicode string concatenation can be achieved using the "+" operator or the StringBuilder class. The "+" operator allows for straightforward concatenation of Unicode strings, treating them as regular strings. However, when concatenating a large number of strings, the "+" operator can lead to performance issues due to the creation of intermediate string objects. To address this, the StringBuilder class provides an efficient solution. It allows for appending Unicode strings without unnecessary object creations, making it suitable for concatenating a significant number of strings or building strings incrementally. By choosing the appropriate approach, Java developers can effectively concatenate Unicode strings while considering performance and efficiency.

5.1. Concatenating Unicode strings using the "+" operator:

- You can concatenate Unicode strings using the "+" operator just like any other strings.

Example 10: Concatenating Unicode string Using the "+" operator:

```
String unicodeStr1 = "ආයුබෝවන්, "; //"Ayubowan" or "Hello" in Sinhala
String unicodeStr2 = "ලෝක!";// World in Sinhala
String concatenatedStr = unicodeStr1 + unicodeStr2;
System.out.println("Concatenated String is: " + concatenatedStr);
//Output:  Concatenated String is: ආයුබෝවන්, ලෝක!
```

5.2. Concatenating Unicode string Using the StringBuilder class:

- The StringBuilder class provides efficient concatenation of strings, including Unicode strings.
- You can use the append() method of StringBuilder to append Unicode strings.
- After appending all the strings, you can convert the StringBuilder back to a regular string using the toString() method.

Example 11: Concatenating Unicode string Using the StringBuilder class:

```
String unicodeStr1 = "ආයුබෝවන්, ";
String unicodeStr2 = "ලෝක!";
StringBuilder stringBuilder = new StringBuilder();
stringBuilder.append(unicodeStr1);
stringBuilder.append(unicodeStr2);
String concatenatedStr = stringBuilder.toString();
System.out.println("Concatenated String is: " + concatenatedStr);
//Output: Concatenated String is: ආයුබෝවන්, ලෝක!
```

6. Introduction to Unicode Charts

Unicode code charts provide a comprehensive visual reference for the characters assigned in the Unicode standard. These charts categorize characters by script, block, or other groupings, making it easier to locate and understand the Unicode code points associated with specific characters. The code charts display the characters, their corresponding code points, and often include additional information such as character names, descriptions, and usage examples.

Unicode code charts serve as a valuable resource for developers, linguists, and anyone working with Unicode characters. They aid in character discovery, understanding character properties, and ensuring accurate character representation across different systems and platforms. By consulting the code charts, users can access a wealth of information about the vast range of characters supported by Unicode, facilitating effective character handling and implementation in software systems.

Example 12: How Unicode Code Charts can be used in Software Development.

Scenario:

Let us say you are developing a text processing application, and you want to implement a feature that converts characters from one script to another. To accomplish this, you need to identify the Unicode code points for the characters in both scripts.

6.1. Accessing Unicode Code Charts:

- Visit the official Unicode website (unicode.org) and navigate to the "Code Charts" section.
- Locate the code chart relevant to the source script you want to convert from and the target script you want to convert to. For example, if you want to convert characters from Latin script to Devanagari script, you would refer to the Latin and Devanagari code charts.

6.2. Finding Unicode Code Points:

- In the code chart, locate the specific characters you need to convert. Each character is represented by its Unicode code point, typically displayed in hexadecimal format.
- Note down the Unicode code points for the source and target characters you want to convert.

6.3. Implementing the Conversion Logic:

- In your software application, use the identified Unicode code points to perform the character conversion.
- Retrieve the source characters from the input text based on their Unicode code points and map them to the corresponding target characters using the desired conversion logic.
- Apply the conversion logic to all relevant characters in the input text, replacing the source characters with the corresponding target characters.

7. Working with Unicode Formats : Storing Unicode Data in UTF-8 Format Files.

7.1. Creating a UTF-8 Encoded File in Java:

The following steps need to be applied in your Java code in order to store Unicode data in a UTF-8 format:

- To store Unicode data in a UTF-8 encoded file in Java, you need to specify the UTF-8 encoding explicitly.
- Create a Writer object, such as BufferedWriter or PrintWriter, and wrap it with an OutputStreamWriter specifying the UTF-8 encoding.
- Open the file for writing and start writing the Unicode text to the file using the Writer object.

Example 13: Storing Unicode Data in a UTF-8 File:

```
import java.io.BufferedWriter;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.io.Writer;

public class UnicodeFileWriter
{
	public static void main(String[] args)
	{
		try (Writer writer = new BufferedWriter(new OutputStreamWriter(new
			FileOutputStream("unicode.txt"), "UTF-8")))
		{
			String unicodeText = "Hello, ලොව!"; // Unicode text to be stored
			writer.write(unicodeText);
		} catch (IOException e)
			{
				e.printStackTrace();
			}
		}
	}
```

7.2. Reading UTF-8 Encoded Files in Java:

The following steps need to be applied in your Java code in order to read UTF-8 encoded file:

- When reading a UTF-8 encoded file that contains Unicode data, you need to ensure that the file is being read using the UTF-8 encoding.
- Create a Reader object, such as BufferedReader or Scanner, and wrap it with an InputStreamReader specifying the UTF-8 encoding.
- Open the file for reading and read the Unicode text from the file using the Reader object.

Example 14: Reading Unicode Data from a UTF-8 File:

```
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.Reader;

public class UnicodeFileReader
{
   public static void main(String[] args)
{
      try (Reader reader = new BufferedReader(new InputStreamReader(new
FileInputStream("unicode.txt"), "UTF-8")))
    {
        StringBuilder sbr = new StringBuilder();
        int c;
        while ((c = reader.read()) != -1) {
           sbr.append((char) c);
     }
        String unicodeText = sbr.toString();
        System.out.println(unicodeText);
     } catch (IOException e) {
        e.printStackTrace();
     }
   }
}
```

7.3. Considerations for Storing Unicode Data in UTF-8 format File:

By following the steps described above, you can store Unicode data in UTF-8 format files and retrieve the data correctly when reading the files in Java. When storing Unicode data in UTF-8 format, it is important to take the following considerations into account:

- Ensure that the file is actually saved with UTF-8 encoding by using appropriate settings in your text editor or IDE.
- Be cautious when manipulating or processing Unicode data, as characters outside the Basic Multilingual Plane (BMP) might require handling surrogate pairs.

Unicode Programming Fundamentals in Python

1. Character as str Data Type in Python

In Python, the str data type represents a sequence of characters. There is no separate char data type; characters are represented as strings of length 1. Since characters are represented as strings, all strings operation and methods can be applied to single characters.

Python offers extensive support for Unicode, allowing developers to work with characters from diverse writing systems. This support is achieved through the use of UTF-8 encoding, which is a variable-length encoding that can represent all possible Unicode characters. Python 3 adopts Unicode as the default encoding for strings, simplifying the handling of international characters and enabling the processing of multilingual text more efficiently[2].

The following Python code demonstrates how char data type can be used in Python to store Unicode characters.

Example 1: Storing Unicode characters in Python:

```
# Define Unicode characters
unicodeChar1 = '\u0627'  # Unicode character for Arabic letter "alef" (ا)
unicodeChar2 = '\u1200'  # Unicode character for Ethiopic letter "ሀ" (ha)
unicodeChar3 = '\u0411'  # Unicode character for  Cyrillic letter 'Б' (be)

# Prints the stored Unicode characters or letters
print("Arabic Letter:", unicodeChar1)  # Output: ا
print("Ethiopic/Ge'ez Letter:", unicodeChar2)  # Output: ሀ
print("Cyrillic Letter:", unicodeChar3)  # Output: Б
```

2. Working with Characters that Fall Outside the BMP in Python

Python provides a range of features and functionalities specifically designed to facilitate working with characters beyond the BMP. These include:

- **Encoding and decoding functions:** Python offers built-in functions that enable the encoding of surrogate pairs into byte sequences and the decoding of byte sequences containing surrogate pairs back into Unicode characters.
- **String operations:** Characters that fall outside the BMP can be treated as individual characters within Python strings, allowing for seamless application of string operations such as concatenation, slicing, and indexing.

Below is a sample code snippet that demonstrates storing and printing the Unicode character that fall outside the BMP in Python.

Example 2: Using Characters that Fall Outside the BMP in Python.

```
# Define the Unicode character using surrogate pair
unicode_char_emoji = "\U0001F60A"

# Prints the smiling face with smiling eyes emoji character
print(unicode_char_emoji)
```

[2] https://docs.python.org/3/howto/unicode.html

3. How to Store and Process Unicode Characters or Strings in Python:

Python's string data type is designed to handle Unicode characters effectively. By using Unicode encoding, literals, encoding/decoding methods, and an extensive range of string operations and methods, developers can work with Unicode characters from various writing systems. The following are key Python features for handling Unicode characters.

String and Bytes Data Type:

- In Python, the string data type (str) is used to store and manipulate Unicode characters.
- The str data type can represent characters from various writing systems, including non-Latin scripts and surrogate pairs or special symbols.
- The bytes and bytearray data types are used to store and manipulate arbitrary bytes as storage units.
- The operation of representing string data (str) as code units (bytes or bytearray) is called *encoding*. The complementary operation, of recovering string data from code units, is called *decoding*.

Unicode Encoding:

- Python 3 defines the string data type as Unicode characters, making it easier to work with international characters and multilingual text.
- When storing Unicode characters in strings, no explicit encoding is required.

Unicode Literals:

- Python supports Unicode literals, which are prefixed with the letter “u”
- Unicode literals allow direct inclusion of Unicode characters in strings without using escape sequences.

Encoding and Decoding

- If you need to convert Unicode strings to byte sequences or vice versa, Python provides methods to encode and decode strings using various encoding schemes, such as UTF-8 and UTF-16.
- The encode() method converts Unicode strings to byte sequences, while the decode() method performs the reverse operation.

String Operations and Methods:

- Python's string operations and methods can be applied to Unicode strings seamlessly.
- String concatenation, slicing, indexing, and searching can handle strings containing Unicode characters from diverse writing systems.

3.1. Unicode string creation:

Example 3: Examples on Unicode String Creation:

- **Directly using Unicode characters :**

  If you cannot switch your keyboard to the script used in the example string, you can use a language translator tool like Google Translate to obtain the string literals for the code snippet.

 # Define the Unicode string variables with string literals

 # The string literals below contain a combination of scripts beyond Latin.
#The non-Latin strings represent the English equivalent of 'World"

unicode_str1 = "Hello, 世界!"

unicode_str2 = "Hello, ዓለም!"

unicode_str3 = "Hello, عالم!"

The accompanying Unicode string for "Hello" represents "World" in Chinese, Ethiopic(Amharic), and Arabic respectively.

- **Using Unicode Escape Sequences:**

#Define Unicode escape sequences for "Hello, ዓለም!"

unicode_str = "\u0048\u0065\u006C\u006C\u006F,
\u12d3\u1208\u121d"

print(unicode_str)

 # prints Hello,ዓለም!

- **Using Unicode Code Points:**

#Define Unicode Code points
code_points = [72, 101, 108, 108, 111, 44, 1593, 1575,1604,1605,33]
unicode_str = "".join(chr(code_point) for code_point in code_points)
print(unicode_str)
 # prints Hello,عالم!

3.2. String length

In Python, you can determine the length of a Unicode string using the len() function. The len() function returns the number of Unicode characters in the string.

Example 4: To Find the Number of Unicode Characters in a Unicode String:

unicode_str = "Hello, عالم!"
length = len(unicode_str)
 # prints Character count is: 12
print("Character counts is: ", length)

3.3. Code Point Count

In Python, the len() function can be used to count the number of Unicode code points in a string.

Example 5: To Find the Number of Code Points in a Unicode String:

unicode_str= "Hello, عالم!"
length = len(unicode_str)
 # prints Code point count is: 12
print("Code point count is : ", length)

3.4. Accessing Individual Code Point

In Python, you can access individual code points in a Unicode string using indexing.

The ord() function is used to access the Unicode code point of a character at a specified indexing of a string. It converts the character to its corresponding Unicode code point.

Example 6: To Find the Code Point at a Given Index:

```
unicode_str = "Hello, عالم"
# Accessing code point at index 0
code_point_0 = ord(unicode_str[0])
print(code_point_0)  # Output: 72
# Accessing code point at index 7
code_point_7 = ord(unicode_str[7])
print(code_point_7)  # Output: 1593
```

Note:

Accessing individual code points by index assumes that the string is well-formed and does not include combining characters or characters that fall outside the BMP. If dealing with more complex Unicode text, it's recommended to use libraries like **unicodedata** or regex for more advanced operations on code points.

4. Iterating Over Code Points

In Python, you can use the ord() function along with a loop control statement to iterate over the code points of a Unicode string.

Example 7: Iterating Individual Code Points within a Unicode String:

```
unicode_str = "Hello, عالم"

for character in unicode_str:

    code_point = ord(character)

    print(code_point)
```

The for loop iterates over each character in the unicode_str. The ord() function is then used to obtain the Unicode code point value for each character, which is printed to the console.

5. Concatenating Unicode Strings

In Python, you can concatenate Unicode strings using the concatenation operator + or the str.join() method.

5.1. Concatenating Unicode Strings Using the + Operator:

- You can concatenate Unicode strings using the "+" operator just like any other strings.

Example 8: Concatenating Unicode strings Using the "+" Operator:

```
unicode_str1 = "Hello, "

unicode_str2 = "عالم"

concatenated_str = unicode_str1 + unicode_str2

print(concatenated_str)

# Output: Hello, عالم
```

5.2. Concatenating Unicode Strings Using the str.join() method:

- The str.join() method can be used to concatenate multiple Unicode strings.

Example 9: Concatenating Unicode string Using the str.join() operator:

```
unicode_strs = ["Hello, ", "عالم, ", "ዓለም"]
concatenated_str = "".join(unicode_strs)
print(concatenated_str)
# Output: Hello, عالم, ዓለም
```

6. Introduction to Unicode Charts

Unicode code charts provide a comprehensive visual reference for the characters assigned in the Unicode standard. These charts categorize characters by script, block, or other groupings, making it easier to locate and understand the Unicode code points associated with specific characters. The code charts display the characters, their corresponding code points, and often include additional information such as character names, descriptions, and usage examples.

Unicode code charts serve as a valuable resource for developers, linguists, and anyone working with Unicode characters. They aid in character discovery, understanding character properties, and ensuring accurate character representation across different systems and platforms. By consulting the code charts, users can access a wealth of information about the vast range of characters supported by Unicode, facilitating effective character handling and implementation in software systems.

Example 12: How Unicode Code Charts can be used in Software Development.

Scenario:

Let us say you are developing a text processing application, and you want to implement a feature that converts characters from one script to another. To accomplish this, you need to identify the Unicode code points for the characters in both scripts.

6.1. Accessing Unicode Code Charts:

- Visit the official Unicode website (unicode.org) and navigate to the "Code Charts" section.
- Locate the code chart relevant to the source script you want to convert from and the target script you want to convert to. For example, if you want to convert characters from Latin script to Devanagari script, you would refer to the Latin and Devanagari code charts.

6.2. Finding Unicode Code Points:

- In the code chart, locate the specific characters you need to convert. Each character is represented by its Unicode code point, typically displayed in hexadecimal format.
- Note down the Unicode code points for the source and target characters you want to convert.

6.3. Implementing the Conversion Logic:

- In your software application, use the identified Unicode code points to perform the character conversion.
- Retrieve the source characters from the input text based on their Unicode code points and map them to the corresponding target characters using the desired conversion logic.
- Apply the conversion logic to all relevant characters in the input text, replacing the source characters with the corresponding target characters.

7. Working with Unicode Transformation Formats : Storing Unicode Data in UTF-8 Format Files.

7.1. Creating a UTF-8 Encoded File in Python:

To store Unicode data in UTF-8 format files in Python, you need to open the file in UTF-8 mode and write the Unicode strings to the file using the specified encoding.

Example 10: Storing Unicode Data in a UTF-8 File:

```
unicode_str = "Hello, عالم"

# Open file in UTF-8 mode for writing

with open("test_file.txt", "w", encoding="utf-8") as file:

    file.write(unicode_str)
```

Note:

- The open() function is used to open a file named "test.txt" in write mode ("w").
- The encoding="utf-8" argument ensures that the file is treated as UTF-8 encoded.
- The write() method is then used to write the Unicode string unicode_string to the file. The string is automatically encoded to UTF-8 before being written.
- By specifying the UTF-8 encoding when opening the file and writing the Unicode string, you ensure that the data is stored in the file in UTF-8 format, which is a widely supported encoding for Unicode.

7.2. Reading UTF-8 Encoded Files in Python:

In Python, to read Unicode data from a UTF-8 formatted file, you can open the file in UTF-8 mode and read the contents using the specified encoding.

Example 11: **Reading Unicode Data from a UTF-8 File:**

```
# Open file in UTF-8 mode for reading
```

```
with open("test.txt", "r", encoding="utf-8") as file:
    file_contents = file.read()
print(file_contents)
```

Note:

- The open() function is used to open a file named "test.txt" in read mode ("r").
- The encoding="utf-8" argument ensures that the file is treated as UTF-8 encoded.
- The read() method is then used to read the contents of the file into the file_contents variable. The contents are automatically decoded from UTF-8 (UTF-8 byte sequences) to Unicode.
- By specifying the UTF-8 encoding when opening the file and reading its contents, you ensure that the data is interpreted correctly as Unicode.

Reference:

1. Unicode Consortium. (n.d.). Home. Retrieved October 3, 2023, from https://home.unicode.org/.
2. Stack Overflow. (n.d.). Where Developers Learn, Share, & Build Careers. Retrieved October 3, 2023, from https://stackoverflow.com/.
3. Python Software Foundation. (n.d.). Unicode HOWTO. Retrieved October 3, 2023, from https://docs.python.org/3/howto/unicode.html.
4. Programming with Unicode. (n.d.). Victor Stinner. Retrieved February 16, 2024, from https://unicodebook.readthedocs.io/.