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

This is the 1st Edition.

1. UA Micro-Learning Module 2: Unicode Advanced Programming.

Welcome to the UA micro-learning module on Advanced Unicode programming!

This module is designed to expand your understanding and proficiency in working with Unicode by covering key aspects such as the character-glyph model, Unicode normalization, accessing the Unicode character database, and comparing Unicode strings. As software development continues to embrace global audiences, it is crucial to master the nuances of Unicode to ensure accurate representation, manipulation, and comparison of text across different languages and scripts. Throughout this module, you will explore the foundations of the character-glyph model, gain insights into text engines that handle complex text layouts, learn about Unicode normalization techniques for ensuring consistency, and explore techniques for accessing the Unicode character database. Additionally, you will discover effective strategies for comparing Unicode strings, accounting for variations in character encoding, diacritics, and other linguistic nuances. The micro-learning module topics covered in this UA curriculum micro-learning module go beyond basic coverage and are driven by region-specific requirements. The following topics are covered to ensure flexible delivery of the micro-learning modules: glyph shapers, text engines and shaped and bidirectional script are covered for flexible delivery of the micro-learning module topics. By the end of this module, you will be equipped with advanced techniques and best practices to work with Unicode data, which enables you to build inclusive and globally accessible software applications.

2. Objectives:

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

- Gain a clear understanding of the character-glyph model and how it relates to Unicode, including the concepts of code points, character encoding, and glyph representations;
- Understand the concept of encoding plain text and the importance of leaving formatting to higher-level protocols, as well as the relationship between encoding and the character-glyph model.
- Explore the functionality and capabilities of text engines that handle complex text layouts, such as bidirectional text, combining characters, and complex scripts;
- Acquire knowledge and practical skills in Unicode normalization techniques, including normalization forms such as NFC (Normalization Form C) and NFD (Normalization Form D).
- Understand the importance of normalization for consistent text processing and comparison;
- Learn how to access and utilize the Unicode character database and and discover how to leverage this database for accurate text handling and manipulation; and
- Understand the challenges and best practices for accurate string comparison in a multilingual and diverse context.

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 aligns with the prerequisite requirements of the Advanced Programming course within existing Information Technology, Computer Science and related programs curricula at higher education institutions, where it is recommended for integration. Additionally, within the intra-dependency of UA micro-learning modules, Module 1-Unicode Programming Fundamentals serves as its prerequisite.

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 **4 hours and 47** minutes. To accommodate linguistic diversities and regional specific requirements in the UA curriculum, the micro-learning module topics are categorized as mandatory and optional. This categorization is necessary because existing standard Computer Science and Information Technology curricula do not include a matching topic for the non-bolded micro-learning module topics. Attempts have been made to include some of these topics, such as Character-glyph model, but only at a high level. The mandatory topics (marked in bold) require **2 hours** and **25 minutes** of classroom time.

Table 1: Course Timing

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 Character-glyph Model.	**8**	**-**	**2**	**10**
Examples: How Character-glyph Model is Implicitly Utilized.	**10**	**5**	**-**	**15**
Text Rendering: Engines, Fonts, and Glyph Shapers.	8	-	5	13
How do glyph shapers handle the shaping of characters with diacritical marks or vowel signs?	8	-	2	10

Normalization of Unicode strings - NFC and NFD.	**10**	**-**	**5**	**15**
Unicode Text Normalization with NFC and NFD in Programming Languages.	**10**	**20**	**-**	**30**
Exploring the Unicode Character Database(UCD) for Better Text Processing.	**10**	**0**	**5**	**15**
How to Access the Unicode Character Database: Methods and Tools.	**10**	**20**	**-**	**30**
Comparing Unicode strings: Case Insensitive and Locale-Based Comparisons.	**10**	**20**	**-**	**30**
Bidirectional scripts and Shaped Scripts.	30	20	-	50
ICU Examples for Complex Script Shaping and Rendering Using Python.	7	8	-	15
Unicode in Other file formats and their handling - JSON File Unicode Handling.	**5**	**5**	**-**	**10**
Unicode String Manipulations Using Language Specific Libraries and ICU.	**10**	**15**	**-**	**25**

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 or course units.

This UA micro-learning module is designed to embrace regional diversities by splitting topic coverage into mandatory and optional topics. The table above lists the micro-learning module topics covered in this micro-learning module. The topics marked in bold are mandatory and deemed sufficient for introducing the Universal Acceptance (UA) curriculum at the academia, while the rest are optional. Educators or trainers can decide which optional topics to cover based on region-specific requirements and the specific needs of their learners.

10. Overview of Character-glyph model

The character-glyph model refers to the fundamental distinction between processing textual information and displaying it visually. In this model, a clear separation is made between characters and glyphs.

Characters represent the abstract units of written language. They are the building blocks of text and carry semantic meaning. Examples of characters include letters, digits, punctuation marks, and symbols. Characters are defined by their Unicode code points, which uniquely identify them.

On the other hand, glyphs are the visual representations or renditions of characters. A glyph is a specific shape or design that corresponds to a character and is used for displaying text

on a screen or in print. Glyphs take into account factors such as font styles, sizes, and specific typographic features.

The key difference between processing textual information and displaying it lies in the distinction between characters and glyphs. When processing textual information, such as manipulating or searching text, the focus is on working with characters at the abstract level. Operations are performed based on the character's Unicode code points, independent of their visual appearance.

In contrast, displaying text involves converting characters into their corresponding glyphs for visual presentation. This process is known as rendering. Rendering considers factors such as font selection, text layout, ligatures, and other typographic aspects to create the visual representation of the text.

It's important to understand this distinction because manipulating text at the character level and manipulating it at the glyph level can have different implications. For example, when performing text operations like sorting or searching, it's essential to consider the character-level properties and behavior rather than relying on the visual appearance of glyphs.

In summary, the character-glyph model emphasizes the separation between the abstract representation of characters and their visual rendering as glyphs, highlighting the distinction between processing textual information and displaying it visually.

11. Examples: How the Character-Glyph Model is Implicitly Utilized.

In both Python and Java, the character-glyph model is implicitly used when working with Unicode strings. Python 3 introduced significant improvements in handling Unicode, making it easier to work with characters and their associated glyphs. Similarly, Java also supports Unicode characters, enabling seamless integration of multilingual text processing within Java applications while adhering to the glyph model.

Example 1: Illustrating How Character-glyph Model is Implicitly Used in Unicode Strings in Python:

```
# Declare a Unicode string
unicode_string = "Hello, 你好, नमस्ते" ##"你好" (nǐ hǎo) is a Chinese greeting, and "नमस्ते" (namaste) is a greeting in Hindi.
# Iterate through the characters in the string
for char in unicode_string:
        print(char)
```

Example 2: Illustrating How Character-glyph Model is Implicitly Used in Unicode Strings in Java:

```
public class UnicodeStringIteration {
  public static void main(String[] args) {
    // Declare a Unicode string
    String unicodeString = "Hello, 你好, नमस्ते"; //"你好" (nǐ hǎo) is a Chinese greeting, and "नमस्ते" (namaste) is a greeting in Hindi.
    // Convert the string to an array of characters
```

```
        char[] charArray = unicodeString.toCharArray();

        // Iterate through the characters in the string
        for (char c : charArray) {
            System.out.println(c);
        }
    }
}
```

In the above example codes, we have a Unicode string that contains a greeting in multiple languages - English, Chinese, and Hindi. By iterating through the characters in the string, we can access and process each individual character, regardless of its specific glyph representation.

Both Python and Java provide support for Unicode, enabling you to perform operations on characters based on their abstract representation, without being concerned about their visual appearance. For example, you can manipulate or compare Unicode characters using their code points, as demonstrated below:

Example 3: Illustrating How Unicode String Comparisons are Performed Using Characters Abstract Representation in Python:

```
# Compare two Unicode characters
char1 = 'ሀ' # the first character (ha) in the Ethiopic script.
char2 = 'ለ' # the character "le" in the Ethiopic script
if ord(char1) < ord(char2):
  print(f"{char1} comes before {char2}")
else:
   print(f"{char1} comes after {char2}")
```

In this code snippet, the ***ord()*** function is used to retrieve the Unicode code point of each character, enabling a comparison based on their abstract representation rather than their appearance as glyphs.

Python also provides various functions and methods for working with Unicode strings, such as encoding and decoding to different character encodings, normalizing Unicode strings, and performing case conversions.

Example 4: Illustrating How Unicode String Comparisons are Performed Using Characters Abstract Representation in Python:

```
public class UnicodeComparison {

   public static void main(String[] args) {

     // Define two Unicode characters using character literals

      char char1 = 'ሀ'; // the first character (ha) in the Ethiopic script
```

```
        char char2 = 'ለ'; // the character "le" in the Ethiopic script


        // Compare the characters directly
        if (char1 < char2) {
            System.out.println(char1 + " comes before " + char2);
        } else {
            System.out.println(char1 + " comes after " + char2);
        }
    }
}
```

By leveraging the character-glyph model in programming languages like Python and Java, developers can work with Unicode characters independently of their visual representation. This enables the development of robust and language-agnostic applications that handle text accurately and consistently across different languages, scripts, and platforms.

12. Text Rendering: Engines, Fonts, and Glyph Shapers

When it comes to displaying text, several essential components work together to ensure accurate and visually appealing rendering. Text engines, fonts, and glyph shapers play crucial roles in this process.

Text Engines:

Text engines are responsible for handling the layout and rendering of text, especially when dealing with complex scripts that involve bidirectional text, combining characters, and complex shaping rules. These engines analyze the Unicode text, apply appropriate layout algorithms, and determine the correct positioning and shaping of characters based on the script's rules. Text engines help ensure proper line breaks, text directionality, and contextual shaping, leading to visually pleasing and linguistically correct text rendering.

Fonts:

Fonts play a crucial role in displaying Unicode text as they provide the necessary glyphs (visual representations) for each character. A font is a collection of glyphs that represent a specific typeface or style. When rendering Unicode text, the font used must support the characters and scripts present in the text. Unicode provides a vast character repertoire, and fonts need to have comprehensive coverage to display characters from various scripts correctly. Choosing the right font ensures that the appropriate glyphs are available for rendering each character in the text.

Glyph Shapers:

Glyph shapers are software components that perform the complex task of shaping glyphs to produce visually connected and contextually appropriate representations of characters. In many scripts, such as Arabic, Indic scripts, or Thai, the shape of a character may vary depending on its position within a word or the surrounding characters. Glyph shapers

analyze the sequence of characters, apply shaping rules specific to the script, and transform the individual glyphs into their appropriate forms. This process ensures that characters are visually connected and displayed correctly according to the script's requirements.

Together, text engines, fonts, and glyph shapers work collaboratively to render Unicode text accurately. Text engines analyze the text and apply layout algorithms, fonts provide the necessary glyphs, and glyph shapers perform shaping operations to create visually cohesive and contextually appropriate representations.

It's important for developers and designers to ensure they use text engines that support the target scripts, select fonts that provide adequate glyph coverage, and utilize glyph shaping libraries or frameworks that understand the shaping rules of various scripts. By leveraging these components effectively, software applications can display Unicode text correctly and provide a seamless reading experience for users across different languages and scripts.

In summary, the following are key points to keep in mind regarding the utilization of text engines, fonts, and glyph shapers**:**

- Displaying text involves the use of text engines, fonts, and glyph shapers.
- Text engines handle tasks such as text layout, line breaking, and shaping.
- Fonts are collections of glyphs that represent the visual variations of characters.
- Fonts specify the specific shape, style, and typographic attributes of glyphs.
- Glyph shapers convert characters into appropriate glyphs based on the chosen font.
- Glyph shaping considers the context and surrounding characters to determine glyph variations and ligatures.
- Text engines, fonts, and glyph shapers work together to ensure accurate and aesthetically pleasing text rendering.
- They handle tasks such as character placement, font selection, ligature handling, and line breaks.
- Understanding these components is important for developers and designers working with text-based applications.
- They enable developers to create visually appealing and readable text displays across different platforms and devices.

13. How do Glyph Shapers Handle the Shaping of Characters with Diacritical Marks or Vowel Signs?

Glyph shapers handle the shaping of characters with diacritical marks or vowel signs by applying specific rules and adjustments to ensure the proper positioning and visual integration of these marks. The following bulleted items describe how glyph shapers handle this task[1]:

- **Base Character and Diacritical Mark Identification:**
  Glyph shapers identify the base character and the associated diacritical mark or vowel sign. The base character is typically shaped first, and then the diacritical mark or vowel sign is positioned relative to it.

- **Positioning and Placement:**
  Glyph shapers determine the correct placement and positioning of diacritical marks

or vowel signs based on the specific script's rules and requirements. The marks need to align properly with the base character, maintaining appropriate spacing and visual harmony. The positioning may differ depending on whether the mark is above, below, before, or after the base character.

- **Mark Positioning Rules:**
  Different scripts have their own rules for positioning diacritical marks. For example, in Arabic script, diacritical marks are often positioned above or below the base character and may have different forms depending on their placement. Glyph shapers analyze these rules and apply the appropriate adjustments to ensure accurate rendering.

- **Contextual Adjustments:**
  Glyph shapers take into account the surrounding characters and adjust the positioning of diacritical marks to maintain proper visual connections and spacing within the word. This includes handling situations where multiple diacritical marks or vowel signs are present in a word and ensuring they are correctly aligned and spaced.

- **Ligatures and Special Forms:**
  In some cases, glyph shapers may substitute a combination of a base character and diacritical mark with a pre-designed ligature or special form. This helps maintain visual consistency and legibility, especially when dealing with complex combinations of characters and diacritical marks.

14. Normalization of Unicode strings - NFC and NFD

Normalization of Unicode strings refers to the process of transforming different representations of the same text into a standardized form. The standardized form is important for operations like testing two strings for equality, collation, storage, and so on. The Unicode standard defines two commonly used normalization forms: NFC (Normalization Form C) and NFD (Normalization Form D).

- **NFC (Normalization Form C):**

NFC is a normalization form where composed characters are used whenever possible. In NFC, characters that can be represented as a combination of a base character and one or more diacritical marks or other combining characters are replaced by their precomposed equivalents. This means that if a character can be represented using a single Unicode code point, it is replaced with that code point. For example, the character "é" (LATIN SMALL LETTER E WITH ACUTE) can be represented as a single precomposed code point (U+00E9) in NFC.

- **NFD (Normalization Form D):**

NFD is a normalization form where characters are decomposed into their base character and combining characters. In NFD, characters that have precomposed forms are decomposed into their constituent parts. This means that if a character can be represented as a combination of a base character and one or more combining characters, it is decomposed into those individual code points. For example, the character "é" (LATIN SMALL LETTER E WITH ACUTE) in NFD is decomposed into the base character "e" (LATIN SMALL LETTER E) with code point U+0065 and

the combining character "´" (COMBINING ACUTE ACCENT) with code point U+0301, where as in NFC, it is represented as a single code point U+00E9.

The purpose of normalization is to ensure text consistency and prevent issues related to canonically equivalent characters. Canonically equivalent characters have different Unicode code points but are considered equivalent in terms of their visual representation. Normalizing text to a specific form helps in comparing, indexing, and searching text accurately.

Understanding and applying Unicode normalization forms like NFC and NFD allows for consistent and reliable handling of text, ensuring proper character composition and decomposition as needed.

Normalization of Unicode strings using NFC (Normalization Form C) and NFD (Normalization Form D) serves the purpose of transforming text into standardized forms.

Let us examine the key aspects and applications of NFC (Normalization Form C) and NFD (Normalization Form D) by comparing their distinctive features:

NFC (Normalization Form C):

- Composes characters and combining marks whenever possible.
- Represents characters as precomposed forms, where a single code point represents a composed character.
- Useful for compatibility with older systems and applications that rely on precomposed characters.
- It is generally preferred for interoperability and compatibility purposes.
- It allows for efficient storage and comparison of strings.
- It can be beneficial when working with most modern software and protocols.

NFD (Normalization Form D):

- Decomposes characters and combining marks.
- Represents characters as a sequence of separate code points, with combining marks following their base characters.
- Useful for applications that require working with decomposed characters and need to accurately process combining marks.
- It can be beneficial for applications that perform in-depth text processing or linguistic analysis.
- It provides a more granular representation of characters, making it suitable for certain text manipulation tasks.
- It can help identify and handle specific linguistic and orthographic variations.

The choice between NFC and NFD largely depends on the specific requirements of the application or system. If compatibility with older systems is crucial, NFC is typically preferred. On the other hand, if working with decomposed characters and combining marks is necessary, NFD is more suitable.

It's important to note that both NFC and NFD help prevent issues related to canonically equivalent characters, ensuring text consistency and accurate processing. By normalizing text to a specific form, developers can avoid problems arising from different Unicode representations of visually equivalent characters.

In summary, NFC is commonly used for compatibility and interoperability purposes, while NFD is favored in cases requiring more granular manipulation of text or linguistic analysis. Understanding the characteristics and use cases of NFC and NFD enables developers to choose the appropriate normalization form for their specific needs.

15. Unicode Text Normalization with NFC and NFD in Programming Languages:

- Prerequisite for running code snippets:

The code snippets(see section 11.1 and 11.2) written below use the Python ***unicodedata*** and the ***java.text.Normalizer*** modules to normalize a Unicode string in NFC (Normalization Form Canonical Composition) and NFD (Normalization Form Canonical Decomposition). There are no specific prerequisites to run this code, but you need to have Python and Java installed on your system.

15.1. Normalization Examples Using Python

In Python, you can use the unicodedata module to normalize Unicode strings to NFC (Normalization Form C) and NFD (Normalization Form D). Below, you will find an illustrative example of NFC and NFD usages:

Example 5: Illustrating How Normalization of Unicode Strings in Python:

```
import unicodedata

# Example text

text = "Café"

# Normalize to NFC

nfc_text = unicodedata.normalize('NFC', text)

print("NFC normalized text:", nfc_text)


#Output: Individual Unicode Code Points for the NFC normalized text: Café

for char in nfc_text:

    if char != text:

        print(char, f"U+{ord(char):04X}")

        # NFC Outputs: diacritic character "é" is U+00E9.


# Normalize to NFD

nfd_text = unicodedata.normalize('NFD', text)

print("NFD normalized text:", nfd_text)

#Output: Individual Unicode Code Points for the NFD normalized text: Café
```

```
for char in nfd_text:
    if char != text:
        print(char, f"U+{ord(char):04X}")
    # NFD Outputs: diacritic character: "e" (U+0065) followed by the combining character "´" (U+0301).
```

In the above code, the unicodedata module provides functions for working with Unicode characters and their properties. A sample text containing the word "Café." is created, then the normalize function from unicodedata to normalize the text to NFC and NFD forms is used.

The normalize function takes two arguments: the normalization form ('NFC' or 'NFD') and the input text. It returns the normalized string according to the specified form.

Example 6: Illustrating How Normalization of Unicode Strings in Python:

```
import java.text.Normalizer;
public class UnicodeNormalization {
    public static void main(String[] args) {
        // Example text
        String text = "Café";
        // Normalize to NFC
        String nfcText = Normalizer.normalize(text, Normalizer.Form.NFC);
        System.out.println("NFC normalized text: " + nfcText);
        // Output: Individual Unicode Code Points for the NFC normalized text: Café
        for (int i = 0; i < nfcText.length(); i++) {
            char ch = nfcText.charAt(i);
            if (ch != text.charAt(i)) {
                System.out.println(ch + " U+" + Integer.toHexString(ch | 0x10000).substring(1));
                // NFC Outputs: diacritic character "é" is U+00E9.
            }
        }

        // Normalize to NFD
        String nfdText = Normalizer.normalize(text, Normalizer.Form.NFD);
        System.out.println("NFD normalized text: " + nfdText);
        // Output: Individual Unicode Code Points for the NFD normalized text: Café
        for (int i = 0; i < nfdText.length(); i++) {
            char ch = nfdText.charAt(i);
            if (ch != text.charAt(i)) {
                System.out.println(ch + " U+" + Integer.toHexString(ch | 0x10000).substring(1));
                  // NFD Outputs: diacritic character: "e" (U+0065) followed by the combining character
"´" (U+0301).
            }
        }
    }
}
```

In both Python and Java examples code given above, oth NFC and NFD normalization produce similar Unicode code point outputs for characters such as C, a, and f. However, there is a difference in the representation of the diacritic character. In NFC, the Unicode code point for "é" is U+00E9, whereas in NFD, it is represented as the base character "e" (U+0065) followed by the combining character "´" (U+0301).

15.2. Normalization Examples Using Java

In Java, you can normalize Unicode strings or texts using NFC and NFD using the **java.text.Normalizer** class. Here's how you can perform normalization using NFC and NFD in Java.

Example 7: Illustrating How Normalization of Unicode Strings in Java:

```
import java.text.Normalizer;

public class UnicodeNormalizationExample {

   public static void main(String[] args) {

      String text = "Café";

      // Normalize to NFC

      String nfcText = Normalizer.normalize(text, Normalizer.Form.NFC);

      System.out.println("NFC normalized text: " + nfcText);

      // Output: Individual Unicode Code Points for the NFC normalized text: Café

      for (int i = 0; i < nfcText.length(); i++) {

         char c = nfcText.charAt(i);

         System.out.printf("%s U+%04X\n", c, (int) c);

      }

      // Normalize to NFD

      String nfdText = Normalizer.normalize(text, Normalizer.Form.NFD);

      System.out.println("NFD normalized text: " + nfdText);


      // Output: Individual Unicode Code Points for the NFD normalized text: Café

      for (int i = 0; i < nfdText.length(); i++) {

         char c = nfdText.charAt(i);

         System.out.printf("%s U+%04X\n", c, (int) c);

      }

   }

}
```

// # NFD Outputs: diacritic character: "e" (U+0065) followed by the combining character "´" (U+0301).

In the example code above, both the normalization forms take a string with a diacritic character, apply normalization to obtain corresponding NFC and NFD normalized texts, and then display the individual Unicode code point values for each character in the normalized texts. The code ensures that combining marks are included in the output by printing each character individually, allowing for a comprehensive examination of the normalization process and the Unicode code points involved.

16. Exploring the Unicode Character Database(UCD) for Better Text Processing:

The Unicode Database is a comprehensive collection of data and information that serves as the backbone for Unicode-aware applications. It contains essential details about every character encoded in the Unicode Standard, including their properties, relationships, and behavior.

The database plays a crucial role in Unicode-aware applications by providing the necessary information to correctly handle, process, and manipulate text and characters from various writing systems, languages, and scripts. It ensures interoperability and consistency in representing and interpreting text across different platforms, devices, and software.

The Unicode Character Database (UCD) is a collection of files that provide essential information and data about Unicode characters. The UCD includes various properties, mappings, and algorithms related to Unicode characters.
The importance of the UCD in Unicode-aware applications can be summarized with the following features it offers:

- **Character Properties and Metadata:** The Unicode Database defines a wide range of properties for each character, such as code point, script, category, decomposition, and many others. These properties enable applications to determine the behavior and characteristics of individual characters, allowing for accurate rendering, sorting, searching, and transformation operations.
- **Collation and Sorting:** The Unicode Database provides collation data, which specifies the order in which characters are sorted in different languages and scripts. This enables applications to perform language-specific sorting and searching, ensuring proper collation of text based on the linguistic rules of specific locales.
- **Normalization and Composition:** Unicode defines normalization forms that ensure text is represented in a consistent manner, regardless of the multiple possible ways characters can be encoded. The Unicode Database includes normalization data, allowing applications to normalize and compose text according to the selected normalization form, ensuring compatibility and consistency in text processing.
- **Bidirectional Text Handling:** The Unicode Database provides rules and data for handling bidirectional text, which is essential for languages that mix left-to-right and right-to-left scripts. It enables applications to accurately display and handle bidirectional text, ensuring proper visual rendering and logical ordering of mixed-script content.
- **Emoji and Emoji Presentation:** The Unicode Database includes detailed information about emojis, such as their properties, sequences, and metadata. This data allows applications to correctly render and handle emojis, including support for emoji presentation variations and skin tone modifiers.

- **Character Set Support and Block Ranges:** The Unicode Database defines the entire Unicode character set and organizes characters into blocks based on their scripts or usage. This information allows applications to determine the supported characters and their ranges, facilitating efficient searching, filtering, and validation of Unicode text.
- **Case Mapping:** The UCD provides case mapping information, including rules for converting characters to uppercase and lowercase forms. This data is used for case-insensitive text matching and transformation.

In summary, the Unicode Database is a vital resource for Unicode-aware applications. Its wealth of information and data enables applications to correctly interpret, process, and display text from diverse languages and scripts, ensuring consistency, interoperability, and accurate handling of Unicode text in a wide range of software systems.

The Unicode Character Database is regularly updated and maintained by the Unicode Consortium, a non-profit organization responsible for the development and maintenance of the Unicode Standard. Software developers and researchers rely on the UCD to handle Unicode characters correctly, perform text processing operations, and ensure interoperability across different platforms and systems.

17. How to Access the Unicode Character Database(UCD): Methods and Tools

There are several methods and tools available for accessing the Unicode Database, providing developers and researchers with different options depending on their specific needs and requirements.

The following are some of the commonly used methods and tools:

- **Programming Language APIs:** Many programming languages provide APIs or libraries that offer direct access to Unicode-related functionalities and data. These APIs often include functions or methods to retrieve character properties, perform text normalization, collation, and other Unicode operations. Examples include the unicodedata module in Python, java.text package in Java, and the ICU library for C/C++.
- **Web-based APIs:** Online web APIs are available that provide access to Unicode data. These APIs allow developers to make HTTP requests to retrieve character properties, search for specific characters, and access other Unicode-related information. The Unicode Consortium offers a web-based API called "Unicode Character Database API" that provides programmatic access to Unicode data.
- **Unicode Database Tools and Utilities:** There are specialized tools and utilities designed to browse, query, and extract information from the Unicode Database. These tools often provide a graphical user interface (GUI) or a command-line interface (CLI) to interact with the database. Examples of such tools include the Unicode Character Database (UCD) Browser, UniView, and ICU tools like ***uconv*** and ***uniset***.
- **Offline Unicode Database Dumps:** Offline dumps of the Unicode Database are available in various formats, such as SQLite databases or custom file formats. These dumps allow for faster access to Unicode data without relying on online resources. Developers can load these dumps into their applications or use them for offline searches and analysis.

- **Unicode Consortium Resources:** The Unicode Consortium, the organization responsible for maintaining the Unicode Standard, provides resources on their website that allow access to Unicode data. These resources include the Unicode Character Database online search, code charts, and related documentation. Developers can utilize these resources to search for character information, explore code points and character properties, and access the latest Unicode releases.
- **Scripts.txt:** Identifying the script from characters used in forming a label using the Scripts.txt file.

It's important to note that the availability of specific tools may vary depending on the programming language, platform, and specific requirements. Developers should choose the method or tool that best aligns with their needs and the capabilities of their programming environment.

In the subsections that follow examples are given to illustrate how to access the Unicode Database using different methods and programming languages.

17.1. Accessing UCD from Programming Languages:

- Prerequisite for running the code snippets:
  - The code snippets(see section 13.1 and 13.2) written below use the Python unicodedata and the java.text.Normalizer modules to normalize a Unicode string in NFC (Normalization Form Canonical Composition) and NFD (Normalization Form Canonical Decomposition). There are no specific prerequisites to run this code, but you need to have Python and Java installed on your system.

Example 8: Examples on illustrating How to Access the Unicode Characters Database Using programming Languages.

- **Python using the unicodedata module:**

```
import unicodedata
char = 'ʊ' # Ethiopic Syllable Ha (ʊ)
category = unicodedata.category(char)
print(f"Character: {char}")
print(f"Category: {category}")
# Output: Character: ʊ
#Output: Category: Lo
```

- **Java using the java.lang.Character package**: the Character class is part of the core Java language and no need to import a package.

```
public class UCDExample2 {
   public static void main(String[] args) {
  // Retrieve character properties
        char character = 'ʊ'; // Ethiopic Syllable Ha (ʊ)
       int codePoint = character;
      int category = Character.getType(codePoint);
     System.out.println("Character: " + character);
     System.out.println("Category: " + category);
     //Output: Character: ʊ
```

```
        //Output: Category: Lo

        }
    }
```

17.2. Using a Command Line Utility or Library:

The ***curl*** command as illustrated in the example below can be used to retrieve valuable information about Unicode characters and their properties from the Unicode character database.
Prerequisite for running the ***curl*** command:

- You must have the curl command-line tool installed on your system. This tool is commonly available on Linux and macOS by default, but you may need to install it on Windows.
- You need an active internet connection to make the HTTP GET request to the specified URL:

  Example 9: Using Command Line Utility

  - ***curl*** *-X GET "https://unicode.org/cldr/utility/character/properties?&character=ʊ"*
  - ***uconv*** -f utf-8 -t ascii input.txt output.txt (converts text from UTF-8 to ASCII)

The ***curl*** command sends a GET request to the Unicode CLDR utility, which provides information about Unicode characters. The Unicode CLDR (Common Locale Data Repository) is a widely used resource for character and locale information.

The above ***curl*** command queries the Unicode character database for information about the character '*ʊ*'. This response is likely to be in JSON or XML format and may include various character properties and data. The response you receive will contain data and properties associated with the specified character, such as its name, category, and other character-related details.

17.3. Using Web Interfaces

The Unicode CLDR utility provides a web interface that allows you to query various character properties. Here's an example of how you can use the Unicode CLDR utility to query the properties of a specific character:

Example 10: Illustrating How to Access the Unicode Characters Database Using Web Interfaces :

- Open your web browser and visit the Unicode CLDR utility website*: https://util.unicode.org/UnicodeJsps/character.jsp*
- In the "Character" field, enter the character for which you want to query properties. For example, let's use the character "*ʊ*".
- Click on the code point of interest.
- The website will display the properties of the character you entered. This includes information such as the character's code point, name, general category, script, block, and various other properties.
- You can scroll down to explore additional details and properties related to the character, such as its decomposition, numeric value, and bidi (bidirectional) properties.

By using the Unicode CLDR utility in this way, you can quickly obtain detailed information about Unicode characters and their properties. It's a convenient tool for exploring and understanding the characteristics and properties of specific characters.

17.4. Illustrating the Applications of Resources from the UCD: Example Script Detection:

The Scripts.txt file can be used for script detection by examining the script property value assigned to each Unicode character. Here are a few examples of how the Script.txt file can be utilized:

Text Analysis: When analyzing a piece of text, each character can be cross-referenced with the Script.txt file to determine its script affiliation. By examining the script property values of consecutive characters, it becomes possible to detect script boundaries and identify which scripts are present in the text.
Language Identification: Many languages are associated with specific scripts. By analyzing the script property values of the characters in a given text, it becomes possible to make educated guesses about the language(s) being used. For example, if a text consists mostly of characters assigned the "Latn" (Latin) script property, it is likely to be in a language written using the Latin script.
User Input Validation: When processing user input, script detection can be used to validate the input against a predefined set of supported scripts. By checking the script property values of the input characters, it becomes possible to ensure that the entered text matches the expected script requirements.

Example 11: Script Detection Using the Scripts.txt File from the UCD:
In order to be able run the Python code below you need to download the Scripts.txt file from the Unicode Character Database.

```
import string
def build_script_map():
    script_map = {}
    with open('Scripts.txt', 'r', encoding='utf-8') as file:
        for line in file:
            line = line.strip()
            if line.startswith('#') or not line:
                continue
            fields = line.split(';')
            char_range = fields[0].strip()
            script = fields[1].strip()
            if '..' in char_range:
                start, end = char_range.split('..')
                start = int(start, 16)
                end = int(end, 16)
                for code_point in range(start, end + 1):
                    char = chr(code_point)
                    script_map[char] = script
            else:
                char = chr(int(char_range, 16))
                script_map[char] = script
    return script_map

def get_script_names(label):
    script_map = build_script_map()
```

```
    seen_scripts = set()
    scripts = []
    label = label.translate(str.maketrans('', '', string.punctuation + '-'))
    for char in label:
        script = script_map.get(char)
        if script is not None and script not in seen_scripts:
            scripts.append(script)
            seen_scripts.add(script)
    return scripts

def extract_script_names(script_strings):
    script_names = []
    seen_scripts = set()
    for script_string in script_strings:
        script_name = script_string.split(' # ')[0].strip()
        if script_name not in seen_scripts:
            script_names.append(script_name)
            seen_scripts.add(script_name)
    return script_names

# Example usage
labels = ["ኢሜይል-መከረ@ሁለንአቀፍ-ተቀባይነት-መከረ.com", "ملموريتانيا", "Կևսսս", "メール例子","exampleمثال"]
for label in labels:
    script_strings = get_script_names(label)
    script_names = extract_script_names(script_strings)
    print(script_names)
```

Output:

```
['Ethiopic', 'Latin']
['Arabic']
['Armenian']
['Bengali', 'Han']
['Latin', 'Arabic']
```

In the above code, the build_script_map() function reads the contents of the 'Scripts.txt' file and constructs a mapping of characters to their associated scripts. It iterates through each line of the file, ignoring comment lines and empty lines. For each valid line, it extracts the character range and script name, and adds the mapping to the script_map dictionary. The get_script_names(label) function takes a label as input and returns a list of script names associated with the characters in the label. It uses the build_script_map() function to obtain the character-to-script mapping. It iterates through each character in the label, checks if it has an associated script in the mapping, and if so, adds the script to the scripts list while ensuring that only the first occurrence of each script is included. The extract_script_names(script_strings) function takes a list of script strings as input and returns a list of unique script names. It iterates through each script string, splits it on the ' # ' delimiter, extracts the script name, and adds it to the script_names list while ensuring that only the first occurrence of each script name is included.

Example 12: Script Detection Using the unicodedata Library in Python:
The example code below demonstrates how to detect the major script categories for different labels after removing punctuation marks and hyphens. This Python code offers a more concise approach to script detection compared to the previously provided version.

```
import unicodedata
import string

def get_script_names(label):
    scripts = set()
    label = label.translate(str.maketrans('', '', string.punctuation + '-'))
    for char in label:
        script = unicodedata.name(char).split()[0]
        scripts.add(script)
    return list(scripts)

# Example usage
label1 = "አሚይል-ሙከራ@ሁለንአቀፍ-ተቀባይነት-ሙከራ.com" # ['LATIN', 'ETHIOPIC']
label2 = "ملموريتانيا" # ['ARABIC']
label3 = "Ընտան" # ['ARMENIAN']
label4 = "মেইল例子" # ['CJK', 'BENGALI']
label5 = "exampleمثال" # ['LATIN', 'ARABIC']

print(get_script_names(label1))
print(get_script_names(label2))
print(get_script_names(label3))
print(get_script_names(label4))
print(get_script_names(label5))
```

Output:

```
['ETHIOPIC', 'LATIN']
['ARABIC']
['ARMENIAN']
['CJK', 'BENGALI']
['ARABIC', 'LATIN']
```

In the above code, the get_script_names() function first removes punctuation marks and hyphens from the label using the translate() method. The function also iterates over each character in the label. For each character, it uses the name method of the unicodedata module to retrieve the Unicode name and splits it to extract the major script category. The major script category is then added to a set of scripts to ensure uniqueness. Finally, the function returns a list of the unique major script categories present in the label.

.

18. Comparing Unicode Strings: Case Insensitive and Locale-Based Comparisons:

When comparing Unicode strings, it's essential to take into account the specific requirements of the comparison. One crucial consideration is whether the comparison should be case-sensitive or

case-insensitive. For case-sensitive comparisons, the individual characters' code points are compared directly. However, for case-insensitive comparisons, additional steps are necessary. It's recommended to use the casefold() method to convert the strings to a case-insensitive form, as it handles various language-specific variations and special characters more effectively than the lower() method. Additionally, normalization can play a role in string comparison. Normalization forms like NFC (Normalization Form C) and NFD (Normalization Form D) can be applied to ensure consistent representation and comparison of composed and decomposed characters. By considering case sensitivity, using appropriate comparison methods, and considering normalization forms, accurate and meaningful comparisons of Unicode strings can be achieved, accounting for the complexities and diversity of characters within the Unicode standard.

When comparing Unicode strings, it is often necessary to consider case sensitivity and locale-based comparisons to ensure accurate and culturally appropriate results.

18.1. Case-insensitive Comparisons:

Case-insensitive comparisons ignore differences in letter case (uppercase or lowercase) when comparing strings. This means that uppercase and lowercase versions of the same letter are considered equivalent.
For example, in a case-insensitive comparison, "apple" and "Apple" would be considered equal, and "hello" and "HELLO" would also be considered equal. Case-insensitive comparisons are useful when you want to perform string comparisons while disregarding differences in letter case.

Python's ***str.casefold()*** method performs case folding, which is a more comprehensive normalization of characters beyond just converting to lowercase. It provides a consistent and reliable way to compare strings while ignoring case distinctions, diacritics, and other similar characters.

Example 13: Implementation of case-insensitive comparison in Python:

```
string1 = "Café"
string2 = "café"
if string1.casefold() == string2.casefold():
        print("The strings are equal (case-insensitive comparison)")
else:
        print("The strings are not equal (case-insensitive comparison)")
#Output: The strings are equal (case-insensitive comparison)
```

Java’s equalsIgnoreCase() method performs case folding identical to the casefold() method of Python.

Example 14: Implementation of case-insensitive comparison in Java:

```
public class CaseInsensitiveComparison {
        public static void main(String[] args) {
         String string1 = "Café";
         String string2 = "café";
```

```
        if (string1.equalsIgnoreCase(string2)) {
            System.out.println("The strings are equal (case-insensitive comparison)");
        } else {
            System.out.println("The strings are not equal (case-insensitive comparison)");
        }
    }
}
```

The above Java code accomplishes the same case-insensitive string comparison as the Python code described above. The equalsIgnoreCase() method is used in Java to perform a case-insensitive comparison of strings.

18.2. Locale-based Comparisons:

Locale-based comparisons take into account the rules and conventions specific to a particular locale or language. Different locales may have different rules for sorting and collation, which determine the order in which strings should be arranged.

For example, in English, "apple" comes before "banana" in a sorting algorithm based on alphabetical order. However, in some other languages, the sorting order may be different due to different alphabets or sorting rules. Locale-based comparisons consider these language-specific rules, allowing you to perform string comparisons that are culturally appropriate.

Python's ***locale.strcoll()*** function performs a locale-based string comparison. It uses the rules defined by the current locale to compare strings, taking into account language-specific sorting and collation. Before using ***locale.strcoll()***, you can set the desired locale using ***locale.setlocale(locale.LC_COLLATE, '<locale>')*** to ensure the correct locale is being used.

Locale-based comparisons are particularly useful when dealing with multilingual text or when you need to sort or compare strings according to language-specific rules. By understanding and utilizing case-insensitive comparisons and locale-based comparisons, you can perform string operations that are more flexible and culturally aware, accommodating different language conventions and user expectations.
Before running the example codes, make sure that the necessary locale is installed.

Example 15: How a Locale-Based Unicode String Comparisons in Python can be Performed Using the *Locale* Module:

```
import locale

# Define the strings to compare

string1 = "café"
string2 = "cafe"
```

```
# Perform a locale-based comparison

def compare_strings(result, locale_name):
    locale.setlocale(locale.LC_ALL, locale_name)
    if result < 0:
        print(f"{string1} comes before {string2} in the {locale_name} locale.")
    elif result > 0:
        print(f"{string1} comes after {string2} in the {locale_name} locale.")
    else:
        print(f"{string1} and {string2} are equivalent in the {locale_name} locale.")

result1 = locale.strcoll(string1, string2)
compare_strings(result1, 'en_US.UTF-8')
result2 = locale.strcoll(string1, string2)
compare_strings(result2, 'fr_FR.utf8')
#First Output: café comes after cafe in the en_US.UTF-8 locale.
#Second Output: café comes before  cafe in the fr_FR.utf8 locale.
```

In the above example, the locale module is used to set the desired locale for comparison, in this case, 'en_US.UTF-8' representing English (United States). Then, we define two Unicode strings, string1 and string2, which are "café" and "cafe" respectively. The **locale.strcoll()** function is used to perform a locale-based comparison between the strings, considering the sorting rules of the specified locale. The result of the comparison is stored in the result variable, which is then checked to determine the relative order of the strings in the specified locale.

Example 16: How a Locale-Based Unicode Strings Comparisons can be Performed in Java Using the *java.text.Collator class:*

```
import java.text.Collator;
import java.util.Locale;

public class UnicodeStringComparison {
    public static void main(String[] args) {
        // Set the desired locale for comparison
        Locale locale = Locale.US; // Example: English (United States)

        // Create a Collator object with the specified locale
        Collator collator = Collator.getInstance(locale);

        // Define the strings to compare
        String string1 = "café";
        String string2 = "cafe";

        // Perform a locale-based comparison
        int result = collator.compare(string1, string2);

        if (result < 0) {
            System.out.println(string1 + " comes before " + string2 + " in the specified locale.");
```

```
        } else if (result > 0) {
            System.out.println(string1 + " comes after " + string2 + " in the specified locale.");
        } else {
            System.out.println(string1 + " and " + string2 + " are equivalent in the specified locale.");
        }
    }
}
//Output: café comes after cafe in the specified locale.
```

Please note that the availability of locales and the specific syntax for setting locales may vary depending on the operating system and programming language environment you are using. Additionally, the locale modules used might require additional configuration or setup on your system. It's recommended to consult the documentation and guidelines specific to your operating system and programming environment for more details on working with locale-based comparisons in the programming language of your choice.

18.3. Locale-Based Sorting

Locale-based sorting, also known as collation or linguistic sorting, refers to the process of sorting strings based on the language-specific rules and conventions of a particular locale. Different languages have different sorting orders and rules for comparing characters.

Example 17: How a Locale-Based Unicode String Sorting can be Performed in Python Using the Locale Module

```
import locale

def sort_word(word, locale_name):
    locale.setlocale(locale.LC_COLLATE, locale_name)
    sorted_word = sorted(word, key=locale.strxfrm)
    return ''.join(sorted_word)

# Example usage
word = "älg"
swedish_sorted_word = sort_word(word, 'sv_SE.UTF-8')
german_sorted_word = sort_word(word, 'de_DE.UTF-8')

    print(f"Swedish sorted characters from the word: {swedish_sorted_word}") # Swedish sorted characters from the word: glä
print(f"German sorted characters from the word: {german_sorted_word}") # German sorted characters from the word: ägl
```

In the above code, the sort_word function takes a word and a locale name as input. It sets the desired locale for comparison using locale.setlocale() with the LC_COLLATE category. Then, it uses the sorted() function with the key parameter set to locale.strxfrm to perform locale-based sorting on the word. The sorted characters are then joined back together to form the sorted word.

19. Bidirectional Scripts and Shaped Scripts

Bidirectional scripts and shaped scripts are concepts related to text rendering and layout, particularly in the context of complex scripts that require special handling due to their inherent characteristics.

19.1. Bidirectional Scripts:

Bidirectional scripts are writing systems in which the text flows from right to left (RTL) for certain scripts (e.g., Arabic, Hebrew, Persian) and left to right (LTR) for other scripts (e.g., English, French, Spanish). The bidirectional nature of these scripts introduces challenges in rendering and handling text correctly, as the order of characters and their visual representation can change depending on the context.

For example, when writing a sentence in Arabic that includes an English word, the English word should appear in its original LTR order within the overall RTL sentence. Proper handling of bidirectional text involves considering the correct order of characters, managing text directionality, and handling bidirectional control characters.

Many modern text rendering engines and frameworks provide support for bidirectional text layout and rendering, allowing developers to handle and display bidirectional text correctly in their applications.

19.2. Shaped Scripts:

Shaped scripts, also known as complex scripts, are writing systems that require additional shaping and contextual transformation of characters to achieve the correct visual representation. These scripts often have complex rules for character joining, ligatures, glyph substitution, and positioning.

Examples of shaped scripts include Arabic, Hebrew, Indic scripts (e.g., Devanagari, Bengali), and Southeast Asian scripts (e.g., Thai, Khmer). In these scripts, characters change their shape depending on their position within a word and the surrounding characters.

Shaping is the process of transforming input text into a sequence of glyphs that visually represent the characters. It involves applying complex algorithms and rules to determine the correct shape of each character in different contexts.

Text rendering engines and libraries often provide shaping engines that handle the complexities of shaped scripts, ensuring accurate rendering of text with appropriate glyph substitutions, ligatures, and positioning.

Supporting bidirectional scripts and shaped scripts requires specialized text rendering engines and libraries that can handle the unique characteristics and complexities of these scripts. These tools ensure that text is rendered correctly, maintaining proper order, directionality, and shaping, to provide a visually accurate representation of the original text.

19.3. Display Format in Bidirectional Texts:

In bidirectional and shaped scripts, the display format plays a crucial role in rendering the text correctly. The display format in these scripts encompasses several significant aspects as described in the sub-sections below.

19.3.1. Bidirectional Display Format:

In bidirectional scripts, the display format considers the inherent bidirectional nature of the text. It ensures that the text flows correctly from right to left (RTL) for RTL scripts and from left to right (LTR) for LTR scripts.

Bidirectional display format involves the following considerations:

- **Text directionality:** The display format determines the base directionality of the text based on the dominant script. It also handles the correct embedding levels and the interaction between RTL and LTR text within the same document or string.
- **Bi-directional control characters:** The display format manages bidirectional control characters that control the text direction, such as the Unicode characters for embedding and overriding the text direction.
- **Contextual analysis:** The format analyzes the context of characters to determine the correct order and positioning of glyphs. It considers factors like script type, character joining, and contextual shaping rules.

Proper implementation of bidirectional display format ensures that bidirectional text is displayed in the correct order and with appropriate directionality, maintaining readability and visual coherence.

19.3.2. Shaped Display Format:

In shaped scripts, the display format handles the complex shaping rules required to render the text accurately. Shaping involves transforming the input text into a sequence of glyphs that represent the characters with the correct visual forms.

Shaped display format involves the following considerations:

- **Character joining:** the format applies rules for joining individual characters to form ligatures and connected forms.
- **Glyph substitution:** the format handles glyph substitutions to display the appropriate form of a character based on its position within a word or its context.
- **Glyph positioning:** the format manages the precise positioning of glyphs to ensure proper spacing and alignment.

Shaped display format ensures that each character appears in its appropriate form, considering its position within a word and the surrounding characters. This results in visually correct representation of shaped scripts, maintaining their unique aesthetic and readability.

Implementing the correct display format for bidirectional and shaped scripts requires specialized text rendering engines, libraries, or frameworks that can handle the complexities of these scripts. These tools manage the text layout, character shaping, and glyph rendering, ensuring that the text is displayed accurately and in a visually pleasing manner.

Example 18: How to Reshape Arabic Text Using the bidi Library in Python:

```
from bidi.algorithm import get_display

def reshape_arabic_text(text):
    reshaped_text = get_display(text)
    return reshaped_text

# Example usage
text = 'مرحبا بكم'  #'Marhaban bikum' in Arabic
reshaped_text = reshape_arabic_text(text)
print(reshaped_text)
```

Output:

```
مكب ابحرم
```

In the code above, the reshape_arabic_text() function uses the get_display() function from the ***bidi.algorithm*** module of the python-bidi library. This function reshapes the Arabic text by applying the Unicode Bidirectional Algorithm, which correctly handles the bidirectional layout of the text.. The output will be the original Arabic text 'مرحبا بكم' in its reshaped form.

Example 19: How to Reshape Arabic Text Using the ICU Library in Java:

```
import com.ibm.icu.text.Bidi;

public class ReshapeArabicText {

   public static String reshapeArabicText(String text) {

      Bidi bidi = new Bidi(text, Bidi.DIRECTION_DEFAULT_LEFT_TO_RIGHT);

      return bidi.writeReordered(Bidi.REORDER_DEFAULT);

   }

   public static void main(String[] args) {

      String text = "مرحبا بكم"; // 'Marhaban bikum' in Arabic

      String reshapedText = reshapeArabicText(text);

      System.out.println(reshapedText);

   }

}
```

The above Java code utilizes the ICU4J library's ***com.ibm.icu.text.Bidi*** package to reshape Arabic text. Within the ReshapeArabicText class, the reshapeArabicText method takes an Arabic text string as input, creates a Bidi object with default left-to-right directionality, and then calls the writeReordered method to obtain the reshaped text according to the default reordering mode.

19.4. File Storage in Key-press Order in Bidirectional and Shaped Scripts:

When storing text in a file in key-press order for bidirectional and shaped scripts, it is important to consider the following aspects:

- **Character Encoding:**

  Ensure that the file is encoded using a character encoding that supports the scripts you are working with. Unicode-based encodings like UTF-8, UTF-16, or UTF-32 are commonly used for storing text that includes bidirectional and shaped scripts. These encodings can represent a wide range of characters and ensure compatibility across different platforms and systems.

- **Logical Order:**

  Bidirectional scripts have a logical order, which is the order in which the characters should be read and processed. For RTL scripts, the logical order is from right to left, and for LTR scripts, it is from left to right. When storing text, make sure to preserve the logical order of the characters.

- **Shaping and Ligatures:**

  Shaped scripts often require special handling for ligatures and contextual shaping.
  When storing text, it is important to store the individual characters in their logical

order, rather than their shaped or ligatured forms. This ensures that the text can be correctly reshaped during rendering.

- **Bi-directional Control Characters:**

Bidirectional scripts use control characters to indicate the directionality of the text. When storing text, keep these control characters intact, as they are important for maintaining the correct display order. Examples of such control characters include the Unicode characters U+202A (Left-to-Right Embedding), U+202B (Right-to-Left Embedding), U+202D (Left-to-Right Override), and U+202E (Right-to-Left Override).

By considering these aspects, you can store text in a file in a way that preserves the key-press order for bidirectional and shaped scripts. It is important to ensure that the file encoding supports the required character set and that the logical order and shaping requirements of the scripts are maintained. This allows the text to be correctly rendered and displayed when it is read from the file and processed by text rendering engines or applications.

19.5. Glyph Shapers in Bidirectional and Shaped Scripts

Glyph shapers play a crucial role in the correct rendering of text in bidirectional and shaped scripts. They are responsible for transforming individual characters into their appropriate glyph forms, considering the script's specific shaping rules and the context of the characters. Let's explore how glyph shapers work in bidirectional and shaped scripts.

19.5.1. Glyph Shaping in Bidirectional Scripts:

In bidirectional scripts, such as Arabic or Hebrew, glyph shapers handle the shaping of individual characters while maintaining the proper order and directionality of the text. Key aspects of glyph shaping in bidirectional scripts include:

- **Contextual Analysis:** Glyph shapers analyze the context of each character, considering surrounding characters, to determine the appropriate glyph forms. This analysis ensures that characters are shaped correctly based on their position within a word and their interaction with neighboring characters.
- **Ligatures and Joining:** Bidirectional scripts often have complex rules for character joining and ligatures. Glyph shapers handle the formation of ligatures and joining of characters, ensuring that the appropriate glyph sequences are generated.
- **Positional Variants:** Some bidirectional scripts, like Arabic, have different glyph forms for initial, medial, final, and isolated positions within a word. Glyph shapers apply the correct positional variants based on the context of each character.

Glyph shapers in bidirectional scripts ensure that the characters are transformed into the correct glyph forms, maintaining the proper shape and visual appearance of the text while respecting the script's inherent bidirectional nature.

19.5.2. Glyph Shaping in Shaped Scripts:

Shaped scripts, such as Arabic, Indic scripts (e.g., Devanagari), or Southeast Asian scripts (e.g., Thai), require extensive glyph shaping due to the variations in character forms based on their position within a word and the surrounding characters. Glyph shaping in shaped scripts involves the following:

- **Contextual Analysis:** Glyph shapers analyze the context of each character, including neighboring characters and script-specific shaping rules, to determine the appropriate glyph

forms. This analysis ensures that characters are shaped correctly based on their position and the script's requirements.

- **Ligatures and Substitutions:** Shaped scripts often involve ligatures and glyph substitutions to form the desired visual representation. Glyph shapers handle these ligatures and substitutions, replacing individual characters with the appropriate glyph sequences.
- **Complex Shaping Rules:** Shaped scripts have intricate shaping rules that govern how characters join, substitute, and transform based on their position and the surrounding characters. Glyph shapers implement these rules to ensure accurate shaping.

Glyph shapers in shaped scripts ensure that the characters are shaped correctly, taking into account the complex shaping rules specific to each script. This results in the visually accurate representation of the text with the appropriate glyph forms and ligatures.

Glyph shapers are typically implemented within text rendering engines or libraries, and developers often rely on these tools to handle the complex shaping process automatically. By utilizing glyph shapers, bidirectional and shaped scripts can be rendered correctly, preserving their visual integrity and ensuring readability.

20. ICU Examples for Complex Script Shaping and Rendering Using Python

ICU (International Components for Unicode) is a powerful library that provides comprehensive support for complex script shaping and rendering. It offers functionalities to handle text layout, shaping, and rendering for various complex scripts, including Indic scripts, Arabic, Thai, and more. Here are a few examples of how you can utilize ICU for complex script shaping and rendering using Python:

Complex script shaping and rendering involve handling scripts that have complex writing systems, such as Arabic, Devanagari, or Thai. ICU provides functionality to shape and render text in complex scripts.

- Prerequisite for running the code snippet:
  - **ICU Library**: The pyicu package is a Python binding for the ICU library. Ensure that the ICU library is installed on your system.
  - **pyicu**: You need to install the pyicu package, which is a Python extension wrapping the ICU library.

Example 20: Shaping and Rendering Arabic Text in Python:

```
import icu

# Create an ICU Transliterator object for Arabic shaping
transliterator = icu.Transliterator.createInstance("Arabic")

# Define the Arabic text to shape and render
arabic_text = "السلام عليكم"  # 'Assalamu alaikum' in Arabic

# Shape the Arabic text
shaped_text = transliterator.transliterate(arabic_text)

# Display the shaped text
```

```
print(shaped_text)
```

Example 21: Shaping and Rendering Indic Script Text (e.g. Devanagari):

```
import icu

# Create an ICU Transliterator object for Devanagari script shaping
devanagari_transliterator = icu.Transliterator.createInstance("Devanagari")

# Define the Devanagari text to shape and render
devanagari_text = "नमस्ते"  # 'Namaste' in Hindi

# Shape the Devanagari text
shaped_text = devanagari_transliterator.transliterate(devanagari_text)

# Display the shaped text
print(shaped_text)
```

Example 22: Shaping and Rendering Thai Text:

```
import icu

# Create an ICU Transliterator object for Thai script shaping
thai_transliterator = icu.Transliterator.createInstance("Thai")

# Define the Thai text to shape and render
thai_text = "สวัสดี"  # 'Sawasdee' in Thai

# Shape the Thai text
shaped_text = thai_transliterator.transliterate(thai_text)

# Display the shaped text
print(shaped_text)
```

These examples demonstrate the basic usage of ICU for complex script shaping and rendering in Python. The ICU library provides more advanced functionalities and options for fine-grained control over text layout, shaping, and rendering. It's recommended to refer to the ICU documentation and API reference for more details on utilizing ICU's capabilities for complex script handling.

Example 23: Shaping and Rendering Arabic Text in Java:

```
import com.ibm.icu.text.ArabicShaping;
import com.ibm.icu.text.Bidi;

public class ArabicTextShaping {
    public static void main(String[] args) {
```

```
        // Create an ICU Arabic text shaping object
        ArabicShaping arabicShaper = new ArabicShaping();

        // Define the Arabic text to shape and render
        String arabicText = "السلام عليكم"; // 'Assalamu alaikum' in Arabic

        // Shape the Arabic text
        String shapedText = arabicShaper.shape(arabicText);

        // Render the shaped text using ICU's BiDi (Bi-Directional) algorithm
        Bidi bidi = new Bidi(shapedText, Bidi.DIRECTION_LEFT_TO_RIGHT);

        // Display the rendered text
        System.out.println(bidi.writeReordered(Bidi.REORDER_DEFAULT));
    }
}
```

The above Java code utilizes ICU4J's ***ArabicShaping*** class to shape Arabic text and Bidi class to apply the BiDi (Bi-Directional) algorithm.

21. Unicode in Other File Formats and their Handling - JSON File Unicode Handling

Unicode is a character encoding standard that aims to represent all written characters and scripts used in modern and historical languages. When working with Unicode in file formats like JSON, it's important to ensure proper handling and representation of Unicode characters.

The following are some considerations for Unicode handling in JSON file handling:

- **Encoding:** JSON files are typically encoded using UTF-8, which is a widely supported Unicode encoding. UTF-8 can represent any Unicode character, ensuring compatibility across different platforms and systems. When creating or reading JSON files, make sure to specify the UTF-8 encoding to properly handle Unicode characters.
- **Escaping Special Characters:** JSON uses backslashes to escape special characters, including Unicode characters that need escaping. For example, characters like newline (\n), tab (\t), or backslash (\) are represented using escape sequences. Unicode characters outside the ASCII range (U+0000 to U+007F) are represented using the "\u" escape sequence followed by the character's Unicode code point in hexadecimal. For example, the character '€' (Euro sign) with code point U+20AC would be represented as "\u20AC" in a JSON string.
- **Unicode Support in JSON Libraries:** JSON libraries and parsers often provide built-in support for Unicode handling, including proper encoding and decoding of Unicode characters. When working with JSON in different programming languages, make sure to use libraries that have Unicode support and handle encoding and decoding correctly. This ensures that Unicode characters are preserved accurately during JSON file handling operations.
- **String Handling:** When working with strings that contain Unicode characters in JSON, ensure that your code or JSON library supports Unicode string handling. The library should

be capable of correctly encoding and decoding Unicode characters to ensure the integrity of the data.

It's important to note that the Unicode handling aspects mentioned here are applicable to JSON files specifically. Other file formats may have their own considerations and requirements for Unicode handling. When working with file formats other than JSON, it's essential to consult the relevant documentation or specifications to understand the recommended practices and guidelines for Unicode handling in those specific formats.

By following proper Unicode handling practices in JSON file handling, you can ensure that Unicode characters are represented accurately and preserved correctly throughout the process of creating, reading, and manipulating JSON files.

Example 24: Unicode Data in JSON File Format in Python:

```
import json
contact_info = {
    "name": {
        "Ethiopic": "ዮናስ ተስፋዬ",
        "Arabic": "يونس تسفاي",
        "Sinhala": "යුනස්ටෙස්ෆායි",
        "Japanese": "ユナス・テスファイ",
        "Chinese": "尤纳斯·特斯法伊",
        "Latin": "Yonas Tesfaye"
    },
   "email_address": {
        "Ethiopic": "ኢሜይል-ሙከራ@ሁለንአቀፍ-ተቀባይነት-ሙከራ.com",
        "Arabic": "تجربة-بريد-الكتروني@تجربة-القبول-الشامل.موريتانيا",
        "Sinhala": "ඉ-තැපෑල-පිරික්සුම@විශ්ව-සම්මුති-පිරික්සුම.ලංකා",
        "Japanese": "ユナス・テスファイ@ユナス・テスファイ",
        "Chinese": "mailto:電子郵件測試@普遍適用測試.台灣",
        "Latin": "yonas.tesfaye@domain.com"
    },
    "job_title": {
        "Ethiopic": "ሶፍትዌር አልሚ",
        "Arabic": "مهندس برمجيات",
        "Sinhala": "මෘදුකාංග ඉංජිනේරු",
        "Japanese": "ソフトウェアエンジニア",
        "Chinese": "软件工程师",
        "Latin": "Software Engineer"
    }
}

#Saving a data to a Json file named contactinfo.json
with open("contactinfo.json", "w", encoding="utf-8") as file:
    json.dump(contact_info, file, ensure_ascii=False)

#Opening a file named "contactinfo.json"
with open("contactinfo.json", "r", encoding="utf-8") as file:
    loaded_data = json.load(file)

# Accessing and printing the loaded data
print(loaded_data["name"])  # John Doe
print(loaded_data["email_address"])  # 30
```

```
print(loaded_data["job_title"])  #こんにちは世界
# Accessing the information in different scripts
print("Name (Ethiopic):", contact_info["name"]["Ethiopic"])
print("Email (Arabic):", contact_info["email_address"]["Arabic"])
print("Job Title (Sinhala):", contact_info["job_title"]["Sinhala"])
print("Name (Japanese):", contact_info["name"]["Japanese"])
print("Email (Chinese):", contact_info["email_address"]["Chinese"])
```

In the above code:

- The ***json.dump()*** function is used to write the data dictionary to a JSON file (data.json). The ensure_ascii=False argument ensures that Unicode characters are not ASCII-encoded and are preserved as-is in the file.
- The **j*son.load()*** function is used to read the data from the JSON file (***data.json***). The loaded data is stored in the loaded_data dictionary.
- Finally, the loaded data is accessed and printed, demonstrating that the Unicode characters are correctly preserved and retrieved from the JSON file.

Note:

- Remember to specify the UTF-8 encoding (encoding="utf-8") when opening the JSON file to ensure proper handling of Unicode characters.
- By using the json module in Python and specifying the appropriate encoding, you can handle Unicode characters in JSON file handling operations.

Example 25: Unicode Data in JSON File Format in Java:

```
import java.io.FileWriter;
import java.io.FileReader;
import java.io.IOException;
import org.json.simple.JSONObject;
import org.json.simple.parser.JSONParser;
import org.json.simple.parser.ParseException;

public class ContactInfo {
    public static void main(String[] args) {
        JSONObject contactInfo = new JSONObject();

        // Define the contact information
        JSONObject name = new JSONObject();
        name.put("Ethiopic", "ዮናስ ተስፋዬ");
        name.put("Arabic", "يونس تسفاي");
        name.put("Sinhala", "යෝනස් ටෙස්ෆායි");
        name.put("Japanese", "ユナス・テスファイ");
        name.put("Chinese", "尤纳斯·特斯法伊");
        name.put("Latin", "Yonas Tesfaye");
        contactInfo.put("name", name);

        JSONObject emailAddress = new JSONObject();
        emailAddress.put("Ethiopic", "ኢሜይል-መከራ@ሁለንአቀፍ-ተቀባይነት-መከራ.com");
```

```
        emailAddress.put("Arabic", "تجربة-بريد-الكتروني@تجربة-القبول-الشامل.موريتانيا");
        emailAddress.put("Sinhala", "ඉ-තැපෑල-පිරික්සුම@විශ්ව-සම්මුති-පිරික්සුම.ලංකා");
        emailAddress.put("Japanese", "ユナス・テスファイ@ユナス・テスファイ");
        emailAddress.put("Chinese", "mailto:電子郵件測試@普遍適用測試.台灣");
        emailAddress.put("Latin", "yonas.tesfaye@domain.com");
        contactInfo.put("email_address", emailAddress);

        JSONObject jobTitle = new JSONObject();
        jobTitle.put("Ethiopic", "ሶፍትዌር አልሚ");
        jobTitle.put("Arabic", "مهندس برمجيات");
        jobTitle.put("Sinhala", "මෘදුකාංග ඉංජිනේරු");
        jobTitle.put("Japanese", "ソフトウェアエンジニア");
        jobTitle.put("Chinese", "软件工程师");
        jobTitle.put("Latin", "Software Engineer");
        contactInfo.put("job_title", jobTitle);

        // Saving data to a JSON file named contactinfo.json
        try (FileWriter file = new FileWriter("contactinfo.json")) {
            file.write(contactInfo.toJSONString());
            System.out.println("Successfully wrote JSON object to file.");
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Opening the file named "contactinfo.json" and accessing the loaded data
        try (FileReader reader = new FileReader("contactinfo.json")) {
            JSONParser jsonParser = new JSONParser();
            JSONObject loadedData = (JSONObject) jsonParser.parse(reader);

            // Accessing and printing the loaded data
            System.out.println("Name: " + loadedData.get("name"));
            System.out.println("Email Address: " + loadedData.get("email_address"));
            System.out.println("Job Title: " + loadedData.get("job_title"));
        } catch (IOException | ParseException e) {
            e.printStackTrace();
        }
    }
}
```

The Java code creates a JSON object representing contact information, including names, email addresses, and job titles in various languages. It then writes this data to a JSON file named "contactinfo.json" using a ***FileWriter***. Afterwards, it reads the saved JSON file using a ***FileReader*** and parses the loaded data into a JSON object. Finally, it accesses and prints the loaded data, demonstrating the process of saving and loading JSON data in Java using the ***org.json.simple*** library.

22. Unicode String Manipulations Using Programming Language Specific Libraries and ICU:

Unicode libraries are essential tools for handling and manipulating text that contains characters from various writing systems and scripts. One popular and widely used Unicode library is the International Components for Unicode (ICU) library. However, there are also other libraries available that provide Unicode support. Let's explore the ICU library and a few other notable programming language specific Unicode libraries.

- **International Components for Unicode (ICU):**
  ICU is a mature and comprehensive library developed by the Unicode Consortium. It offers extensive Unicode and globalization support for software applications. ICU provides functions and classes for character handling, string manipulation, collation, formatting, and more. It supports a wide range of programming languages, including C/C++, Java, and Python.
- **Java-specific Unicode Libraries:**
  - ***java.text:*** The ***java.text*** package in Java provides several classes for working with Unicode text, such as ***Collator*** for string comparison, ***Normalizer*** for Unicode normalization, and **BreakIterator** for text segmentation.
  - ***icu4j:*** ICU for Java (icu4j) is the Java version of the ICU library. It offers extensive Unicode and globalization support similar to the ICU C/C++ library.
- **Python-specific Unicode Libraries:**
  - ***unicodedata***: the ***unicodedata*** module is part of Python's standard library and provides access to the Unicode Character Database. It allows you to retrieve information about Unicode characters, such as their names, categories, and properties.
  - ***PyICU:*** is a Python version of the ICU library. It offers extensive Unicode and globalization support similar to the ICU C/C++ library.
  - **regex**: the **regex** library is an alternative to Python's built-in **re** module that provides enhanced Unicode support. It supports various Unicode properties, character classes, and grapheme cluster segmentation.

These are just a few examples of Unicode libraries available for the two programming languages. Depending on your specific requirements and programming language, you can choose the library that best suits your needs for working with Unicode text.

When working with Unicode in Java and Python, you can leverage Unicode libraries like **ICU** (International Components for Unicode) and other Unicode-related libraries to handle various Unicode operations.

Example 26: Unicode String Manipulations in Python Using the ICU Library:

```
import icu

# Define the Japanese text to shape and render
japanese_text = "こんにちは世界"  # 'Hello, World' in Japanese

# Create an ICU Transliterator object for Han-Latin script shaping
han_latin_transliterator = icu.Transliterator.createInstance("Han-Latin")
```

```
# Shape the Japanese text
shaped_text = han_latin_transliterator.transliterate(japanese_text)

# Display the shaped text
print(shaped_text)

# Character count
character_count = icu.UnicodeString(japanese_text).length()
print("Character count:", character_count)

# Character iteration and properties
unicode_string = icu.UnicodeString(japanese_text)
for i in range(character_count):
    code_point = unicode_string.charAt(i)
    char = chr(code_point)
    char_name = icu.Char.charName(code_point)
    is_digit = str(char).isdigit()
    is_uppercase = str(char).isupper()
    print("Character:", char)
    print("Character code point:", code_point)
    print("Character name:", char_name)
    print("Is character a digit?", is_digit)
    print("Is character uppercase?", is_uppercase)
    print("--------------------")
```

The above Python code demonstrates how to use the ***icu*** module to iterate over the characters in a Unicode string. It retrieves the character count, creates a UnicodeString object, the text is then shaped using the **transliterate()** method of the Transliterator object, and then iterates over each character. For each character, it obtains the character's code point, converts it to a character, retrieves the character's name, and checks if it is a digit or uppercase. In summary, the code showcases how to access various properties of Unicode characters using the **icu** module.

Apart from ICU, there are other Unicode-related libraries available in Python, such as **unicodedata**, which provides access to Unicode character properties, and regex, which offers advanced regular expressions with Unicode support.

By utilizing Unicode libraries like **ICU** and others, you can perform advanced Unicode operations, handle character properties, normalization, and more in your Python code.

22.1. Examples on How to Use Programming Language Specific Libraries for String Processing:

The examples below demonstrate how to perform Unicode string manipulations, such as normalization and removing diacritical marks, using language-specific libraries (***java.text.Normalizer*** in Java and ***unicodedata, regex*** *libraries* in Python) as well as ICU (International Components for Unicode) versions of Java and Python. The normalization process ensures that the strings are in a standardized form, while removing diacritical marks allows working with accent-insensitive text.

Prerequisites to run the codes:

- Java examples:

- For the Java ICU example, you need to install the icu4j library or include the library in your project's dependencies.
- Python examples:
  - For the Python ICU example, you need to install the PyICU package
  - For the Python regex library example, you need to have the regex library installed.

Example 27: Unicode String Manipulations in Java using Built-in Library:

```
import java.util.Iterator;
public class UnicodeExampleICU {
    public static void main(String[] args) {
        // Example Unicode string
        String text = "こんにちは世界"; // Japanese greeting "Hello, World"
        // Character count
        int characterCount = text.codePointCount(0, text.length());
        System.out.println("Character count: " + characterCount);
        // Character iteration and properties
        Iterator<Integer> codePointIterator = text.codePoints().iterator();
        while (codePointIterator.hasNext()) {
            int codePoint = codePointIterator.next();
            System.out.println("Character: " + (char) codePoint);
            System.out.println("Character code point: " + codePoint);
            System.out.println("Character name: " + Character.getName(codePoint));
            System.out.println("Is character a digit? " + Character.isDigit(codePoint));
            System.out.println("Is character uppercase? " + Character.isUpperCase(codePoint));
            System.out.println("--------------------");
        }
    }
}
```

Note:

- The import icu statement is not needed in Java, as the Unicode Character Database is already part of the Java standard library.

Example 28: Java - Using the *java.text.Normalizer* class for Unicode Normalization:

```
import java.text.Normalizer;
public class UnicodeStringManipulationJava {
    public static void main(String[] args) {
        String input = "Café";

        // Normalize the string to NFC form
        String normalized = Normalizer.normalize(input, Normalizer.Form.NFC);

        // Remove diacritical marks
        String withoutDiacritics = normalized.replaceAll("\\p{M}", "");

        System.out.println("Normalized: " + normalized);
        System.out.println("Without Diacritics: " + withoutDiacritics);
    }
}
```

Example 29: Java - Using ICU (icu4j) for Unicode Normalization and Case Folding:

```
import com.ibm.icu.text.Normalizer2;
import com.ibm.icu.text.Transliterator;

public class UnicodeStringManipulationICUJava {
   public static void main(String[] args) {
      String input = "Café";

      // Normalize the string to NFC form using ICU
      Normalizer2 normalizer = Normalizer2.getNFCInstance();
      String normalized = normalizer.normalize(input);

      // Remove diacritical marks using ICU
      Transliterator diacriticRemover = Transliterator.getInstance("Any-NFD; [:M:] Remove;
     NFC");
      String withoutDiacritics = diacriticRemover.transform(normalized);

      System.out.println("Normalized: " + normalized);
      System.out.println("Without Diacritics: " + withoutDiacritics);
   }
}
```

Example 30: Python - Using the unicodedata Module for Unicode Normalization and Character properties:

```
import unicodedata

input = "Café"

# Normalize the string to NFC form
normalized = unicodedata.normalize('NFC', input)

# Remove diacritical marks
without_diacritics = ''.join(c for c in normalized if not unicodedata.combining(c))

print("Normalized:", normalized)
print("Without Diacritics:", without_diacritics)
```

Example 31: Python - Using the PyICU library for Unicode Normalization and Case Folding:

```
import PyICU
input = "مَرْحَبًا" # 'Marhaban' in Arabic

# Normalize the string to NFC form using PyICU
normalizer = PyICU.Normalizer('nfc')
normalized = normalizer.normalize(input)

# Remove diacritical marks using PyICU
diacritic_remover = PyICU.Transliterator.createInstance('Any-NFD; [:M:] Remove; NFC')
without_diacritics = diacritic_remover.transliterate(normalized)

print("Normalized:", normalized)
print("Without Diacritics:", without_diacritics)
```

Output

Normalized: مَرْحَبًا

Without Diacritics: مرحبا

Example 32: Python - Using the regex Library for Unicode Normalization and Case Folding:

```
import regex
import unicodedata
input_str = "مَرْحَبًا" #‘Marhaban’ in Arabic
# Normalize the string to NFC form
normalized = unicodedata.normalize('NFC', input_str)
# Remove diacritical marks using regex
removed_diacritics = regex.sub(r'\p{M}', '', normalized)

print("Original:", input_str)
print("Normalized:", normalized)
print("Without Diacritics:", removed_diacritics)
```

Output

Original: مَرْحَبًا

Normalized: مَرْحَبًا

Without Diacritics: مرحبا

In the code above, the regex library is used for Unicode-aware regular expressions. The normalize() function is used to normalize the input string to the NFC (Normalization Form Canonical Composition) form. The sub() function is then used with the pattern r'\p{M}' to match and remove all combining marks (diacritics) from the string.

Both the Java and Python codes presented above produces identical output such as the following- the normalized string and the string without diacritics are printed:

Normalized: Café

Without Diacritics: Cafe

22.2. Examples on How to Use Third Party Library (ICU) in Python for Text Normalization and Collation:

Text Normalization:

- Text normalization is the process of transforming text into a canonical or normalized form. ICU provides normalization functions based on Unicode normalization forms.

Example 33: Third ICU Library in Python for Text Normalization:

```
import icu
def normalize_text(text):
    normalizer = icu.Normalizer2.getNFCInstance()
    normalized_text = normalizer.normalize(text)
    return normalized_text
text = "تفاحة"
normalized_text = normalize_text(text)
print("Normalized Text:", normalized_text)
```

Output:
Normalized Text: تفاحة

In the above example, the ***icu.Normalizer*** class is used to normalize the text to NFC(Normalization Form C). The resulting normalized text is then printed.

Text Collation:

Text collation is the process of comparing and sorting text based on linguistic rules. ICU provides collation functionality to compare and sort text according to various locales and collation rules.

Example 34: Generic ICU Library in Python for Text Collation:

```
import icu
def compare_strings(text1, text2, locale='ar'):
    collator = icu.Collator.createInstance(icu.Locale(locale))
    result = collator.compare(text1, text2)
    if result < 0:
        return f"{text1} comes before {text2}"
    elif result > 0:
        return f"{text1} comes after {text2}"
    else:
        return f"{text1} is equal to {text2}"

string1 = "تفاحة" # 'tuffaha' in Arabic, Apple in English
string2 = "موز" # 'mawz' in Arabic, Banana in English
comparison = compare_strings(string1, string2)
print("Comparison Result:", comparison)
```

Output:
Comparison Result: تفاحة ***comes before*** موز

In the above Python example code, the ***icu.Collator*** class is used to create a collator instance for the specified locale (default is 'en_US'). The compare() method is then used to compare the two strings (text1 and text2), and the result is returned.You can also customize the collation behavior by specifying locale-specific collation rules or using different collation strengths (e.g., primary, secondary, or tertiary) for different levels of comparison.
These examples demonstrate how to use ICU in Python for text normalization and collation. The **icu** module provides comprehensive support for various text processing tasks, and you can explore the ICU documentation for more details on the available functions, options, and customization for text normalization and collation based on your specific requirements.

Example 35: ICU Library in Java for Text Collation:

```
import com.ibm.icu.text.Collator;
import com.ibm.icu.util.ULocale;
public class StringComparison {
    public static String compareStrings(String text1, String text2, String locale) {
        Collator collator = Collator.getInstance(new ULocale(locale));
        int result = collator.compare(text1, text2);
        if (result < 0) {
            return text1 + " comes before " + text2;
        } else if (result > 0) {
            return text1 + " comes after " + text2;
```

```
        } else {
            return text1 + " is equal to " + text2;
        }
    }
    public static void main(String[] args) {
        String string1 = "تفاحة"; // 'tuffaha' in Arabic, Apple in English
        String string2 = "موز"; // 'mawz' in Arabic, Banana in English
        String comparison = compareStrings(string1, string2, "ar");
        System.out.println("Comparison Result: " + comparison);
    }
}
```

Reference:

1. Unicode Consortium. "Glyph Shaping." In The Unicode Standard, Version 15.0. Unicode Consortium. 2022. Retrieved October 21, 2023, from https://www.unicode.org/versions/Unicode15.0.0/.
2. Python Software Foundation. (2023). Python 3.12 Documentation. Retrieved October 21, 2023, from https://docs.python.org/3/.
3. International Business Machines Corporation (IBM). (2023). ICU Documentation. Retrieved October 21, 2023, from https://docs.python.org/3/.