What is Unicode and how does it works?

What is Unicode: A Universal Character Set

Unicode is a standard that assigns a unique number to every character, no matter the platform, program, or language. It's like a global dictionary for characters.

Why is it important?

Before Unicode, different systems used different character sets, leading to compatibility issues. For instance, a document created on one system might appear garbled when opened on another. Unicode solved this problem by providing a unified standard.

Key points about Unicode:

- **Comprehensive:** It covers almost all writing systems in the world, including Latin, Cyrillic, Greek, Arabic, Chinese, Japanese, Korean, and many more.
- **Unique identifier:** Each character has a unique code point, a number that represents it.
- **Encoding formats:** Unicode characters are often stored in different encoding formats like UTF-8, UTF-16, and UTF-32.

In essence, Unicode is the foundation for consistent text representation across different systems and applications.

UTF-8 can represent how many characters?

UTF-8 doesn't have a fixed number of characters. The key point to understand is that UTF-8 is an encoding scheme, not a character set. It's a way to represent characters from the Unicode standard in a binary format.

How UTF-8 Works

- **Variable-length encoding:** UTF-8 uses 1 to 4 bytes to represent a character. This flexibility allows it to encode a vast range of characters, from basic ASCII to complex symbols.
- **Compatibility:** It's backward compatible with ASCII, meaning ASCII characters are represented by a single byte.

Why There's No Fixed Number

The number of characters in a UTF-8 encoded text depends entirely on the specific characters used. For example:

- A text containing only ASCII characters will use one byte per character.
- A text with a mix of Latin characters and emojis might use a combination of one, two, or three-byte sequences.

In Summary

Instead of focusing on the total number of characters in UTF-8, it's more practical to consider:

- **The range of characters:** UTF-8 can represent over 1 million characters.
- **The byte size of individual characters:** This varies from one to four bytes.
- **The specific text:** The actual character count depends on the content.

Can you explain UTF-8 Encoding with and example?

The Sentence:

Let's take the sentence: "नमस्ते, Hello! How are you? 😊 "

Breaking it Down:

- **नमस्ते:** This is a Hindi word. In UTF-8, it will likely be represented using multiple bytes for each character, as Hindi characters often require more than one byte.
- **Hello!:** These English characters will be represented using a single byte each, as they fall within the ASCII range.
- **How are you?:** Similar to "Hello!", these English words will use one byte per character.
- 😊**:** This emoji is a special character and will likely be encoded using multiple bytes.

Encoding Process:

1. Each character is converted into its corresponding Unicode code point.
2. The Unicode code point is then converted into a UTF-8 byte sequence based on specific encoding rules.

Example (Simplified):

- **न:** Might be encoded as a three-byte sequence like `11100110 10101001 10110010`
- **म:** Might be encoded as a three-byte sequence like `11100110 10101010 10110101`
- **स:** Might be encoded as a three-byte sequence like `11100110 10101100 10110111`
- **्त:** Might be encoded as a three-byte sequence like `11100110 10101101 10111000`
- **े:** Might be encoded as a three-byte sequence like `11100110 10101110 10111001`
- **,** will be encoded as a single byte `00101100`
- **H:** will be encoded as a single byte `01001000`
- **e:** will be encoded as a single byte `01100101`
- **l:** will be encoded as a single byte `01101100`
- **l:** will be encoded as a single byte `01101100`
- **o:** will be encoded as a single byte `01101111`
- **!:** will be encoded as a single byte `00100001`
- **:** will be encoded as a single byte `00111010`
- **H:** will be encoded as a single byte `01001000`
- **o:** will be encoded as a single byte `01101111`
- **w:** will be encoded as a single byte `01110111`
- ... (similar encoding for the rest of the sentence)

- **😊:** Might be encoded as a four-byte sequence (specific bytes will depend on the emoji's Unicode code point)

Note: This is a simplified explanation and the actual encoding might be different. The purpose is to illustrate the concept of variable-length encoding in UTF-8.

Why UTF-8 is called 8 and not 4 or 10?

UTF-8 stands for Unicode Transformation Format - 8 bit.

The "8" in UTF-8 refers to the fact that it uses **8-bit bytes** as the fundamental unit for encoding characters. This means each individual byte in a UTF-8 sequence consists of 8 bits.

The Misconception

The confusion might arise because:

- **Variable-length encoding:** UTF-8 uses a variable number of bytes (1, 2, 3, or 4) to represent characters. This might give the impression it's dealing with 4-byte units.
- **Other encodings:** There are other Unicode encodings like UTF-16 and UTF-32 which use 16-bit and 32-bit units respectively.

Key Point

While UTF-8 can use up to 4 bytes to represent a single character, the underlying unit it operates on is still an 8-bit byte. This is why it's called UTF-8.

To summarize:

- UTF-8 uses 8-bit bytes as its building blocks.
- It's a variable-length encoding, meaning characters can be represented using 1 to 4 bytes.
- The "8" in UTF-8 refers to the byte size, not the character size.

What are different unicoding systems

While UTF-8 has become the dominant character encoding, there are several others that have been used historically or are still in use for specific purposes. Let's explore some of them:

Legacy Encoding Systems

- **ASCII (American Standard Code for Information Interchange):** This was one of the earliest encoding systems, representing 128 characters (including uppercase and lowercase letters, numbers, punctuation, and control characters) using 7 bits.
- **ISO-8859:** A family of character encodings based on the Latin alphabet, each covering different character sets for specific regions (e.g., Western Europe, Latin America, Cyrillic).
- **EBCDIC (Extended Binary Coded Decimal Interchange Code):** Primarily used in IBM mainframe computers, EBCDIC differs significantly from ASCII in character representation.

Unicode Encodings (Apart from UTF-8)

- **UTF-16:** Uses 16-bit code units to represent characters. It's more efficient for languages with a large character set, but can be less efficient for English text compared to UTF-8.
- **UTF-32:** Uses 32-bit code units for each character, providing a fixed-width representation. While simple to implement, it can be inefficient in terms of storage space.

Other Notable Encodings

- **GB2312, GBK, GB18030:** These are Chinese character encoding standards used in mainland China.
- **Shift-JIS, EUC-JP:** Japanese character encodings.
- **Big5:** Traditional Chinese character encoding.

Choosing the Right Encoding

The best encoding for a specific task depends on several factors:

- **Character set:** The range of characters you need to represent.
- **Efficiency:** The desired balance between storage space and processing speed.
- **Compatibility:** The systems and applications that will handle the encoded data.

UTF-8 has gained popularity due to its flexibility, compatibility, and efficiency in handling a wide range of characters. However, understanding other encoding systems is essential for working with legacy data or specific character sets.

How computer/system/human can know from the sequence of words that next character is 1 byte or 2 bytes or 3 bytes or 4 bytes long?

How UTF-8 Determines Character Length

UTF-8 employs a clever system to indicate the length of a character based on the pattern of bits in the first byte.

The Byte Pattern:

- **Single-byte characters:** If the first byte is in the range 0x00 to 0x7F (00000000 to 01111111), it represents a single-byte character (ASCII).
- **Multi-byte characters:** If the first byte starts with 0b110 (110xxxxx), it's the beginning of a two-byte character.
- If the first byte starts with 0b1110 (1110xxxx), it's the beginning of a three-byte character.
- If the first byte starts with 0b11110 (11110xxx), it's the beginning of a four-byte character.

Continuation Bytes:

Following bytes in a multi-byte character always start with 0b10 (10xxxxxx). This pattern helps identify the continuation bytes of a character.

Example:

Let's say we have the following byte sequence: `11000010 10111101 11100110 10011100 10101010 10111101`

- The first byte `11000010` starts with `110`, indicating a two-byte character.
- The second byte `10111101` starts with `10`, confirming it's a continuation byte for the first character.
- The third byte `11100110` starts with `1110`, indicating a three-byte character.
- The fourth, fifth, and sixth bytes start with `10`, confirming they are continuation bytes for the third character.

Summary:

By examining the first byte of a character sequence, a computer can determine the number of bytes required to represent the character. The subsequent bytes, with their specific pattern, confirm the character's length. This mechanism ensures accurate decoding of UTF-8 text.

UTF-8, UTF 16, UTF 32 all can represent same number of characters. Then why to chose UTF 32?

The key difference lies in their efficiency and specific use cases.

When to Consider UTF-32:

While UTF-32 might seem like the straightforward choice due to its fixed-width nature, it's generally not the preferred encoding for most applications. However, there are specific scenarios where it can be advantageous:

1. **Simplicity and Performance:** If you prioritize simplicity in character handling and have ample storage space, UTF-32 can be attractive. Basic text manipulation operations might be slightly faster due to the fixed-width nature.
2. **Internal Representations:** Some software applications might use UTF-32 internally for processing text, even if the external data is stored in a different encoding. This can simplify character handling within the application.
3. **Specific Use Cases:** In rare cases where every character needs to be accessed independently without considering byte boundaries or character lengths, UTF-32 might be suitable.

Why UTF-8 is Often Preferred:

- **Efficiency:** UTF-8 is generally more space-efficient, especially for text containing primarily ASCII characters.
- **Wide Compatibility:** It's supported by almost all systems and applications.
- **Flexibility:** Handles a wide range of character sets efficiently.

When to Consider UTF-16:

- **Legacy Systems:** Some older systems might still use UTF-16, so compatibility might be a factor.
- **Balancing Efficiency and Complexity:** It offers a balance between UTF-8 and UTF-32 in terms of space efficiency and complexity.

In conclusion, while UTF-32 can be a viable option in specific circumstances, UTF-8 is generally the preferred encoding due to its efficiency, compatibility, and flexibility.

Author
Dr Hari Thapliyaal
dasarpai.com
linkedin.com/in/harithapliyal