# The Move Book

While Move does not have a built-in type to represent strings, it does have two standard implementations for strings in the Standard Library . The std::string module defines a String type and methods for UTF-8 encoded strings, and the second module, std::ascii , provides an ASCII String type and its methods.

The Sui execution environment automatically converts bytevector into String in transaction inputs. As a result, in many cases, constructing a String directly within the Transaction Block is unnecessary.

No matter which type of string you use, it is important to know that strings are just bytes. The wrappers provided by the string and ascii modules are just that: wrappers. They do provide safety checks and methods to work with strings, but at the end of the day, they are just vectors of bytes.

While there are two types of strings ( string and ascii ) in the standard library, the string module should be considered the default. It has native implementations of many common operations, leveraging low-level, optimized runtime code for superior performance. In contrast, the ascii module is fully implemented in Move, relying on higher-level abstractions and making it less suitable for performance-critical tasks.

The String type in the std::string module is defined as follows:

To create a new UTF-8 String instance, you can use the string::utf8 method. The Standard Library provides an alias .to_string() on the vector for convenience.

UTF8 String provides a number of methods to work with strings. The most common operations on strings are: concatenation, slicing, and getting the length. Additionally, for custom string operations, the bytes() method can be used to get the underlying byte vector.

The default utf8 method may abort if the bytes passed into it are not valid UTF-8. If you are not sure that the bytes you are passing are valid, you should use the try_utf8 method instead. It returns an Option , which contains no value if the bytes are not valid UTF-8, and a string otherwise.

Hint: Functions with names starting with try_* typically return an Option . If the operation succeeds, the result is wrapped in Some . If it fails, the function returns None . This naming convention, commonly used in Move, is inspired by Rust.

The string module does not provide a way to access individual characters in a string. This is because UTF-8 is a variable-length encoding, and the length of a character can be anywhere from 1 to 4 bytes. Similarly, the length() method returns the number of bytes in the string, not the number of characters.

However, methods like sub_string and insert validate character boundaries and abort if the specified index falls within the middle of a character.

This section is coming soon!

## Strings are bytes

No matter which type of string you use, it is important to know that strings are just bytes. The wrappers provided by the string and ascii modules are just that: wrappers. They do provide safety checks and methods to work with strings, but at the end of the day, they are just vectors of bytes.

```bash module book::custom_string;

/// Anyone can implement a custom string-like type by wrapping a vector. public struct MyString { bytes: vector, }

/// Implement a `from_bytes` function to convert a vector of bytes to a string. public fun from_bytes(bytes: vector): MyString { MyString { bytes } }

/// Implement a `bytes` function to convert a string to a vector of bytes. public fun bytes(self: &MyString): &vector { &self.bytes } ```

While there are two types of strings ( string and ascii ) in the standard library, the string module should be considered the default. It has native implementations of many common operations, leveraging low-level, optimized runtime code for superior performance. In contrast, the ascii module is fully implemented in Move, relying on higher-level abstractions and making it less suitable for performance-critical tasks.
```

The String type in the std::string module is defined as follows:

`bash // File: move-stdlib/sources/string.move /// A `String` holds a sequence of bytes which is guaranteed to be in utf8 format. public struct String has copy, drop, store { bytes: vector<u8>, }`

To create a new UTF-8 String instance, you can use the string::utf8 method. The Standard Library provides an alias .to_string() on the vector for convenience.

``bash // the module isstd::stringand the type isString` use std::string::{Self, String};

// strings are normally created using the `utf8` function // type declaration is not necessary, we put it here for clarity let hello: String = string::utf8(b"Hello");

// The `.to_string()` alias on the `vector<u8>` is more convenient let hello = b"Hello".to_string(); ```

UTF8 String provides a number of methods to work with strings. The most common operations on strings are: concatenation, slicing, and getting the length. Additionally, for custom string operations, the bytes() method can be used to get the underlying byte vector.

```bash let mut str = b"Hello,".to_string(); let another = b" World!".to_string();

// append(String) adds the content to the end of the string str.append(another);

// `sub_string(start, end)` copies a slice of the string str.sub_string(0, 5); // "Hello"

// `length()` returns the number of bytes in the string str.length(); // 12 (bytes)

// methods can also be chained! Get a length of a substring str.sub_string(0, 5).length(); // 5 (bytes)

// whether the string is empty str.is_empty(); // false

// get the underlying byte vector for custom operations let bytes: &vector = str.bytes(); ```

The default utf8 method may abort if the bytes passed into it are not valid UTF-8. If you are not sure that the bytes you are passing are valid, you should use the try_utf8 method instead. It returns an Option , which contains no value if the bytes are not valid UTF-8, and a string otherwise.

Hint: Functions with names starting with try_* typically return an Option . If the operation succeeds, the result is wrapped in Some . If it fails, the function returns None . This naming convention, commonly used in Move, is inspired by Rust.

```bash // this is a valid UTF-8 string let hello = b"Hello".try_to_string();

assert!(hello.is_some()); // abort if the value is not valid UTF-8

// this is not a valid UTF-8 string let invalid = b"\xFF".try_to_string();

assert!(invalid.is_none()); // abort if the value is valid UTF-8 ```

The string module does not provide a way to access individual characters in a string. This is because UTF-8 is a variable-length encoding, and the length of a character can be anywhere from 1 to 4 bytes. Similarly, the length() method returns the number of bytes in the string, not the number of characters.

However, methods like sub_string and insert validate character boundaries and abort if the specified index falls within the middle of a character.

This section is coming soon!

## Working with UTF-8 Strings

While there are two types of strings ( string and ascii ) in the standard library, the string module should be considered the default. It has native implementations of many common operations, leveraging low-level, optimized runtime code for superior performance. In contrast, the ascii module is fully implemented in Move, relying on higher-level abstractions and making it less suitable for performance-critical tasks.

The String type in the std::string module is defined as follows:

`bash // File: move-stdlib/sources/string.move /// A `String` holds a sequence of bytes which is guaranteed to be in utf8 format. public struct String has copy, drop, store { bytes: vector<u8>, }

To create a new UTF-8 String instance, you can use the string::utf8 method. The Standard Library provides an alias .to_string() on the vector for convenience.

``bash // the module isstd::stringand the type isString` use std::string::{Self, String};

// strings are normally created using the `utf8` function // type declaration is not necessary, we put it here for clarity let hello: String = string::utf8(b"Hello");

// The `.to_string()` alias on the `vector<u8>` is more convenient let hello = b"Hello".to_string(); ```

UTF8 String provides a number of methods to work with strings. The most common operations on strings are: concatenation, slicing, and getting the length. Additionally, for custom string operations, the bytes() method can be used to get the underlying byte vector.

```bash let mut str = b"Hello,".to_string(); let another = b" World!".to_string();

// append(String) adds the content to the end of the string str.append(another);

// `sub_string(start, end)` copies a slice of the string str.sub_string(0, 5); // "Hello"

// `length()` returns the number of bytes in the string str.length(); // 12 (bytes)

// methods can also be chained! Get a length of a substring str.sub_string(0, 5).length(); // 5 (bytes)

// whether the string is empty str.is_empty(); // false

// get the underlying byte vector for custom operations let bytes: &vector = str.bytes(); ```

The default utf8 method may abort if the bytes passed into it are not valid UTF-8. If you are not sure that the bytes you are passing are valid, you should use the try_utf8 method instead. It returns an Option , which contains no value if the bytes are not valid UTF-8, and a string otherwise.

Hint: Functions with names starting with try_* typically return an Option . If the operation succeeds, the result is wrapped in Some . If it fails, the function returns None . This naming convention, commonly used in Move, is inspired by Rust.

```bash // this is a valid UTF-8 string let hello = b"Hello".try_to_string();

assert!(hello.is_some()); // abort if the value is not valid UTF-8

// this is not a valid UTF-8 string let invalid = b"\xFF".try_to_string();

assert!(invalid.is_none()); // abort if the value is valid UTF-8 ```

The string module does not provide a way to access individual characters in a string. This is because UTF-8 is a variable-length encoding, and the length of a character can be anywhere from 1 to 4 bytes. Similarly, the length() method returns the number of bytes in the string, not the number of characters.

However, methods like sub_string and insert validate character boundaries and abort if the specified index falls within the middle of a character.

This section is coming soon!

## ASCII Strings

This section is coming soon!