BString Class Documentation

Overview

`BString` is a high-performance, mutable container for holding an ordered sequence of strings. ⛓ It is written as a Python C extension to provide maximum speed for data-heavy tasks, while offering a rich, Pythonic API that feels like a supercharged list.

It is designed for efficient data manipulation, transformation, and serialization to and from various formats. `BString` is a core component of the `BeautifulString` library.

Features

* **High-Performance**: Core logic is written in C for speed, especially for bulk operations and file I/O.
* **Full Mutability**: Supports `.append()`, `.insert()`, `.pop()`, `.remove()`, `.extend()`, and item assignment.
* **Complete Slicing**: Full `list`-like slice support for getting, setting, and deleting.
* **Rich Data Export**: Convert instances on-the-fly to `list`, `tuple`, `dict`, `JSON`, and highly configurable `CSV` formats.
* **Advanced File I/O**: Robust methods for reading/writing text files and multi-line `CSV` files with header support.
* **Powerful Transformations**: Includes `.map()`, `.filter()`, and a `.transform_chars()` method for bulk character removal/retention.
* **Intuitive Navigation**: An explicit cursor model with `.head`, `.tail`, `.current`, and methods like `.move_next()` to iterate without creating new objects.
* **Content Querying**: Includes `.contains()` for substring searches and `.unique()` for deduplication.

API and Usage

Initialization

A `BString` object is initialized with zero or more strings as positional arguments.

```python
from BeautifulString import BString

# Initialize with multiple strings
fruits = BString("apple", "banana", "cherry")
print(fruits)
# Output: ['apple', 'banana', 'cherry']

# Initialize an empty BString
empty_b = BString()
print(empty_b)
# Output: []
```

Navigation Properties & Methods

BString uses an explicit cursor model for navigation.

Properties

- **.head** (read-only str): Returns the **first** string in the sequence. Does not move the cursor.
- **.tail** (read-only str): Returns the **last** string in the sequence. Does not move the cursor.
- **.current** (read-only str): Returns the string at the **current position** of the internal cursor.

```
fruits = BString("apple", "banana", "cherry")
print(f"Head: {fruits.head}")    # Output: Head: apple
print(f"Tail: {fruits.tail}")    # Output: Tail: cherry
```

print(f"Current: {fruits.current}") # Output: Current: apple

Methods

- **.move_next()**: Advances the internal cursor one step forward. Returns True if successful, False if at the end.
- **.move_prev()**: Moves the internal cursor one step backward. Returns True if successful, False if at the beginning.
- **.move_to_head()**: Resets the internal cursor to the first element.
- **.move_to_tail()**: Moves the internal cursor to the last element.

fruits = BString("apple", "banana", "cherry")

print(f"Initial: {fruits.current}") # Output: Initial: apple

fruits.move_next()

print(f"After move_next(): {fruits.current}") # Output: After move_next(): banana

fruits.move_to_tail()

print(f"After move_to_tail(): {fruits.current}") # Output: After move_to_tail(): cherry

Mutability Methods

- **.append(value)**: Adds a string to the end of the sequence.
- b = BString('a')
- b.append('b')
- print(b) # Output: ['a', 'b']

- **.extend(iterable)**: Appends all strings from an iterable to the end.
- b = BString('a')
- b.extend(['b', 'c'])
- print(b) # Output: ['a', 'b', 'c']

- **.insert(index, value)**: Inserts a string at a specific position.
- b = BString('a', 'c')

- b.insert(1, 'b')
- print(b) # Output: ['a', 'b', 'c']

- **.pop([index])**: Removes and returns the string at index (default: last).
- b = BString('a', 'b', 'c')
- last = b.pop()
- print(f"Popped '{last}', BString is now {b}")
- # Output: Popped 'c', BString is now ['a', 'b']

- **.remove(value)**: Removes the first occurrence of a string. Raises ValueError if not found.
- b = BString('a', 'b', 'a')
- b.remove('a')
- print(b) # Output: ['b', 'a']

Slicing and Operators

Full slicing and standard sequence operators are supported.

```
b = BString('a', 'b', 'c', 'd', 'e')

# Get a slice (returns a new BString)
print(b[1:4]) # Output: ['b', 'c', 'd']

# Set a slice
b[1:3] = BString('X', 'Y', 'Z')
print(b) # Output: ['a', 'X', 'Y', 'Z', 'd', 'e']

# Operators
b_sum = BString('x') + BString('y')
```

b_mul = BString('z') * 2

print(b_sum) # Output: ['x', 'y']

print(b_mul) # Output: ['z', 'z']

Transformation & Querying Methods

- **.map(method_name, *args)**: Applies a string method to every element, returning a new BString.
- b = BString(' apple ', ' pear ')
- print(b.map('strip')) # Output: ['apple', 'pear']

- **.filter(condition, *args)**: Returns a new BString containing elements that satisfy a condition (either a string method or a callable).
- b = BString('Apple', 'Pear', 'GRAPES')
- print(b.filter('startswith', 'P')) # Output: ['Pear']
- print(b.filter(lambda s: s.isupper())) # Output: ['GRAPES']

- **.transform_chars(characters, mode='remove', inplace=False)**: Removes or keeps a specific set of characters in each string.
- b = BString("Order: #123-A")
- print(b.transform_chars("0123456789", mode='keep')) # Output: ['123']

- **.contains(substring, case_sensitive=True)**: Returns True if any string in the BString contains the substring.
- b = BString("Helsinki", "Rovaniemi, Lapland")
- print(b.contains('lapland', case_sensitive=False)) # Output: True

- **.unique()**: Returns a new BString with duplicate strings removed, preserving order.
- b = BString("apple", "banana", "apple")

- print(b.unique()) # Output: ['apple', 'banana']

- **.join(separator)**: Joins all elements into a single Python string.
- b = BString("Rovaniemi", "Lapland", "Finland")
- print(b.join(", ")) # Output: Rovaniemi, Lapland, Finland

- **BString.split(string, delimiter=None)** (Class Method): Creates a BString by splitting a string.
- log_line = "INFO;user-login;lehto"
- b = BString.split(log_line, delimiter=";")
- print(b) # Output: ['INFO', 'user-login', 'lehto']

Data Export (__call__)

Call the BString instance like a function to convert it to various formats.

- **container='list'**:
- b = BString('a', 'b')
- print(b(container='list')) # Output: ['a', 'b']

- **container='tuple'**:
- b = BString('a', 'b')
- print(b(container='tuple')) # Output: ('a', 'b')

- **container='dict'**:
- b = BString('Alice', 'Finland')
- print(b(container='dict', keys=['name', 'country']))
- # Output: {'name': 'Alice', 'country': 'Finland'}

- **container='json'**:

- b = BString('user', 'rovaniemi')
- print(b(container='json')) # Output: ["user", "rovaniemi"]
- print(b(container='json', keys=['name', 'city'])) # Output: {"name": "user", "city": "rovaniemi"}

- **container='csv'**:
- b = BString('user with, comma', 'rovaniemi')
- print(b(container='csv', delimiter='|')) # Output: "user with, comma"|rovaniemi

File I/O

BString includes powerful methods for handling text and CSV files.

- **.to_file(filepath)** & **BString.from_file(filepath)**:
- b = BString("First line.", "Second line.")
- b.to_file("lines.txt")
- loaded_b = BString.from_file("lines.txt")
- print(loaded_b) # Output: ['First line.', 'Second line.']
- # os.remove("lines.txt")

- **BString.to_csv(...)** & **BString.from_csv(...)**:
- header = BString("ID", "Name")
- data_rows = [BString("1", "Alice"), BString("2", "Bob")]
- BString.to_csv("data.csv", data=data_rows, header=header)
-
- loaded_header, loaded_data = BString.from_csv("data.csv", header=True)
- print(f"Header: {loaded_header}, Data: {loaded_data[0]}")
- # Output: Header: ['ID', 'Name'], Data: ['1', 'Alice']
- # os.remove("data.csv")