4. String Operations

1. Creating Strings

In Python, you can create strings by enclosing characters within single quotes (`'`), double quotes (`"`), or triple quotes (`'''` or `"""`). Here are some examples:

```python
string1 = 'Hello, World!'
string2 = "This is another string."
string3 = '''This is a
multi-line
string.'''
```

2. String Indexing and Slicing

Each character in a string has an index, which represents its position within the string. Indexing in Python starts from 0, so the first character has an index of 0, the second character has an index of 1, and so on.

You can access individual characters using square brackets `[]` and the index position. For example:

```python
string = "Python"
print(string[0])  # Output: 'P'
print(string[3])  # Output: 'h'
```

Slicing allows you to extract a substring from a string by specifying a range of indices. The syntax is `string[start:end:step]`, where `start` is the index to start from (inclusive), `end` is the index to stop at (exclusive), and `step` is the optional step size.

```python
string = "Hello, World!"
print(string[0:5])  # Output: 'Hello'
print(string[7:12])  # Output: 'World'
print(string[::2])  # Output: 'Hlo ol!'
print(string[::-1])  # Output: '!dlroW ,olleH'
```

3. String Concatenation

Concatenation is the operation of combining two or more strings into a single string. In Python, you can use the `+` operator to concatenate strings.

```
string1 = "Hello, "
string2 = "World!"
combined_string = string1 + string2
print(combined_string)  # Output: 'Hello, World!'
```

4. String Multiplication

In Python, you can multiply a string by an integer to create a new string that repeats the original string a specified number of times.

```
string = "Python "
repeated_string = string * 3
print(repeated_string)  # Output: 'Python Python Python '
```

5. String Methods

Python provides a wide range of built-in methods for working with strings. Here are some commonly used string methods:

- `string.upper()`: Returns a new string with all characters converted to uppercase.
- `string.lower()`: Returns a new string with all characters converted to lowercase.
- `string.strip()`: Returns a new string with leading and trailing whitespace removed.
- `string.split(separator)`: Splits the string into a list of substrings based on the specified separator.
- `string.replace(old, new)`: Returns a new string with all occurrences of the `old` substring replaced by the `new` substring.
- `string.join(iterable)`: Joins the elements of an iterable (e.g., a list) into a single string, using the string as the separator.

```
string = "   Hello, World!   "
print(string.upper())  # Output: '   HELLO, WORLD!   '
print(string.lower())  # Output: '   hello, world!   '
print(string.strip())  # Output: 'Hello, World!'

words = string.split(",")
print(words)  # Output: ['   Hello', ' World!   ']
```

```
new_string = string.replace("World", "Python")
print(new_string)  # Output: '   Hello, Python!   '

list_of_words = ["Python", "is", "awesome"]
joined_string = " ".join(list_of_words)
print(joined_string)  # Output: 'Python is awesome'
```

6. String Formatting

Python provides various ways to format strings, allowing you to insert values into placeholders within a string. One common method is the `format()` method, which uses curly braces `{}` as placeholders.

```
name = "Alice"
age = 25
formatted_string = "My name is {} and I'm {} years old.".format(name,
age)
print(formatted_string)  # Output: 'My name is Alice and I'm 25 years
old.'
```

You can also use f-strings (formatted string literals) introduced in Python 3.6, which provide a more concise and readable way to format strings.

```
name = "Bob"
age = 30
formatted_string = f"My name is {name} and I'm {age} years old."
print(formatted_string)  # Output: 'My name is Bob and I'm 30 years
old.'
```

See Also

5. Inserting variables into strings