String

A **string** is a sequence of characters enclosed within quotes in Python. It can be created using either **single quotes (' ')** or **double quotes (" ")**.

For example, the following are strings:

"Hello, World!"
'Python is awesome!'
"123"
'@#$%^&*()'

Strings are considered as a fundamental **data type** in Python programming and are used extensively in various programming tasks, such as data input/output, data manipulation, and web development.

Strings are **immutable**, meaning that once a string is created, it cannot be modified. However, you can create a new string by concatenating or slicing the original string.

String Operation

Here is a table of common string operations in Python along with a brief description:

Operation	Description
`+`	Concatenation: combines two or more strings into a single string.
`*`	Repetition: repeats a string a specified number of times.
`[]`	Indexing: returns the character at the specified index.
`[:]`	Slicing: returns a substring of the original string.
`In`	Membership: checks if a substring is present in the original string.
`not in`	Membership: checks if a substring is not present in the original string.
`%`	Formatting: formats a string with variables.
`.format()`	Formatting: formats a string with variables.

+ operator

The **+ operator** is used for concatenation of two or more strings. It combines the strings on either side of the operator and returns a new string.

Example:

```
greeting = "Hello"
name = "Alice"
message = greeting + ", " + name
print(message)

# Output: Hello, Alice
```

*** operator**

The *** operator** is used for repetition of a string. It repeats the string on the left side of the operator by the number of times specified on the right side of the operator.

Example:

```
greeting = "Hello"
message = greeting * 3
print(message)

# Output: HelloHelloHello
```

% operator

The **% operator** is used for string formatting. It allows you to insert values into a string using placeholders.

Example:

```
age = 25
message = "I am %d years old" % age
print(message)

# Output: I am 25 years old
```

in operator

The **in operator** is used to check if a substring is present in a string. It returns `True` if the substring is found and `False` otherwise.

Example:

```
my_string = "Python is awesome"
print("is" in my_string)

# Output: True
```

indexing string

In Python, indexing is used to access individual characters in a string using their position (index). The index of a character in a string starts at 0 for the first character, 1 for the second character, and so on.

You can use square brackets `[]` to access a particular character at a specific index position.

Example:

```
my_string = "Hello, World!"
print(my_string[0])

# Output: H
```

In the above example, my_string[0] returns the first character of the string, which is 'H'.

You can also use negative indexing to access characters from the end of the string. The index -1 refers to the last character of the string, -2 refers to the second last character, and so on.

Example:

```
my_string = "Hello, World!"
print(my_string[-1])

# Output: !
```

In the above example, my_string[-1] returns the last character of the string, which is '!'.

If you try to access an index that is outside the range of the string, Python will raise an `IndexError` exception.

Example:

```
my_string = "Hello, World!"
print(my_string[20])

# Raises an IndexError exception
```

In the above example, `my_string[20]` tries to access an index that is outside the range of the string, so Python raises an `IndexError` exception.

In summary, indexing is a fundamental operation for working with strings in Python. It allows you to access individual characters in a string, which is useful for many programming tasks, such as data manipulation and text processing.

Slicing

In Python, **slicing** is used to extract a part of a string by specifying the **start** and **end** index. Slicing is done using the square bracket notation **[start:end]**. The `start` index is inclusive, and the `end` index is exclusive, which means that the character at the `end` index is not included in the sliced string.

Example:

```
my_string = "Hello, World!"
print(my_string[0:5])

# Output: Hello
```

In the above example, **my_string[0:5]** extracts the characters from index 0 up to index 5 (excluding index 5), which gives the substring "Hello".

You can also **omit the start** index to start from the beginning of the string, or omit the `end` index to slice until the end of the string.

Example:

```
my_string = "Hello, World!"
print(my_string[:5]) # Output: Hello
print(my_string[7:]) # Output: World!
```

In the above examples, **my_string[:5]** extracts the characters from the beginning of the string up to index 5 (excluding index 5), which gives the substring "Hello". **my_string[7:]** extracts the characters from index 7 until the end of the string, which gives the substring "World!".

You can also use **negative indices in slicing** to count from the end of the string.

Example:

```
my_string = "Hello, World!"
print(my_string[-6:-1]) # Output: World
```

In the above example, **my_string[-6:-1]** extracts the characters from index -6 up to index -1 (excluding index -1), which gives the substring "World".

Example program that demonstrates string slicing in Python:

```
my_string = "The quick brown fox jumps over the lazy dog"

# Print the first word
print(my_string[:3]) # Output: The

# Print the last word
print(my_string[-3:]) # Output: dog

# Print every other character
print(my_string[::2]) # Output: Teqikbonfxjmsoe h aydg

# Print the string in reverse order
print(my_string[::-1]) # Output: g od yzal eht revo spmuj xof nworb kciuq ehT
```

In the above program, we define a string **my_string** and then demonstrate several ways to slice the string using indexing.

- **my_string[:3]** slices the string from the beginning up to (but not including) the 3rd index, which gives us the first word "The".
- **my_string[-3:]** slices the string from the 3rd character from the end (inclusive) to the end of the string, which gives us the last word "dog".
- **my_string[::2]** slices the string with a step of 2, which means it includes every other character in the string.
- **my_string[::-1]** slices the string with a step of -1, which means it includes all characters in reverse order.

format()

In Python, the **format()** method is used to format a string by replacing placeholders with values. The basic syntax of the **format()** method is:

formatted_string = "string with placeholders {}".format(value)

In the above syntax, the curly braces **{}** represent a placeholder in the string, which can be replaced with a value using the **format()** method.

Example

```
name = "John"
age = 30
formatted_string = "My name is {} and I am {} years old".format(name, age)
print(formatted_string) # Output: My name is John and I am 30 years old
```

In the above example, the string "My name is {} and I am {} years old" contains two placeholders {} which are replaced with the values of the name and age variables using the format() method.

You can also use indexed placeholders in the string to specify the order in which the values should be inserted.

Example:

```
name = "John"
age = 30
formatted_string = "My name is {0} and I am {1} years old. {0} is my first name.".format(name, age)
print(formatted_string)

# Output: My name is John and I am 30 years old. John is my first name.
```

In the above example, the string "My name is **{0}** and I am **{1}** years old. **{0}** is my first name." contains two indexed placeholders **{0}** and **{1}** which are replaced with the values of the name and age variables in the order specified.

You can also use named placeholders in the string to provide more descriptive names for the values.

Example:

```
person = {"name": "John", "age": 30}  # Dictionary
formatted_string = "My name is {name} and I am {age} years old".format(**person)
print(formatted_string)

# Output: My name is John and I am 30 years old
```

In the above example, a dictionary person is used to store the values of the name and age variables. The **format()** method is called with the double-asterisk syntax ****person**, which unpacks the dictionary and uses its key-value pairs as named arguments for the placeholders in the string.

Escape Sequencing

When you have a string that contains both single and double quotes, you may encounter a **SyntaxError** if you try to print the string using the same type of quotes that are already present in the string. This is because Python interprets the closing quote within the string as the end of the string itself, leading to a syntax error.

Example:

```
str = "They asked, "what's going on in class?""

print(str)
```

Output:

```
SyntaxError: invalid syntax
```

To print such a above string, you have a couple of options

1. **Triple Quotes**: You can use triple quotes (`'''` or `"""`) to enclose the string. Triple quotes allow you to include both single and double quotes within the string without causing a syntax error. Example:

   ```
   print('''"They asked, "what's going on in class?""''')
   ```

   Output:

   ```
   "They asked, "what's going on in class?""
   ```

2. **Escape Sequences**: Alternatively, you can use escape sequences (`\'` or `\"`) to escape the quotes within the string. By preceding the quote with a backslash, you indicate that it should be treated as a literal character, rather than as the end of the string.

Example:

```
print("\"They asked, \"what\'s going on in class?\"\"")
```

Output:

'They asked, "what's going on in class?""

List of an Escape Sequence Characters

1. \n (Newline): This escape sequence possessed the power to create a new line within a string. When encountered, it waved its wand and gracefully inserted a line break, allowing text to flow onto the next line. Example:

```
print("Once upon a time,\nthere was a brave knight.")
```

Output:

```
Once upon a time,
there was a brave knight.
```

2. \t (Tab): Known as the trickster of escape sequences, this character had the ability to summon horizontal tabs. It would swiftly create empty spaces that aligned text in a neat and organized manner. Example:

```
print("Name:\tArthur")
print("Age:\t25")
```

Output:

```
Name:   Arthur
Age:    25
```

3. \ (Backslash): The guardian of backslashes, this escape sequence ensured their safe passage through strings. It cleverly escaped itself to protect other characters from misinterpretation. Example:

```
print("Path: C:\\Program Files\\Python")
```

Output:

```
Path: C:\Program Files\Python
```

4. " (Double Quote): This escape sequence possessed the ability to encase a string within double quotes. It provided a shield of protection, allowing the string to retain its original form. Example:

```
print("She said, \"Hello!\"")
```

Output:

```
She said, "Hello!"
```

5. ' (Single Quote): A close companion of strings enclosed in single quotes, this escape sequence enabled the inclusion of single quotes within such strings. It ensured that the beauty of the quotes remained unharmed. Example:

```
print('He said, \'Greetings!\'')
```

Output:

```
He said, 'Greetings!'
```

String Handling Functions

In the below table of some commonly used string handling functions in Python, along with their descriptions given:

Function	Description
`len(string)`	Returns the length of the string
`string.lower()`	Returns a copy of the string with all uppercase characters converted to lowercase
`string.upper()`	Returns a copy of the string with all lowercase characters converted to uppercase
`string.strip()`	Returns a copy of the string with leading and trailing whitespace removed
`string.lstrip()`	Stripping: removes whitespace or specified characters from the beginning of a string.
`string.rstrip()`	Stripping: removes whitespace or specified characters from the end of a string.
`string.split(separator)`	Returns a list of substrings separated by the specified `separator`
`string.count()`	Counting: returns the number of occurrences of a substring in a string.
`string.index()`	Searching: returns the index of the first occurrence of a substring in a string.
`string.find(substring)`	Returns the index of the first occurrence of `substring` in the string, or `-1` if not found

Function	Description
`string.replace(old, new)`	Returns a copy of the string with all occurrences of `old` replaced with `new`
`string.startswith(prefix)`	Returns `True` if the string starts with `prefix`, otherwise `False`
`string.endswith(suffix)`	Returns `True` if the string ends with `suffix`, otherwise `False`
`string.isdigit()`	Returns `True` if all characters in the string are digits, otherwise `False`
`string.isalpha()`	Returns `True` if all characters in the string are alphabetic, otherwise `False`
`string.isalnum()`	Returns `True` if all characters in the string are alphanumeric, otherwise `False`
`string.islower()`	Returns `True` if all characters in the string are lowercase, otherwise `False`
`string.isupper()`	Returns `True` if all characters in the string are uppercase, otherwise `False`
`string.capitalize()`	Returns a copy of the string with the first character capitalized and the rest in lowercase
`string.join(iterable)`	Returns a new string that is the concatenation of the strings in the specified iterable, separated by the original string.
int(string)	This function converts a string to an integer. If the string contains non-numeric characters, a ValueError will be raised.
str(number)	This function converts a number to a string.
chr(number)	The chr() function is useful when you need to convert an integer code point to its corresponding character, particularly when working with Unicode characters and encoding/decoding operations.
ord(character)	The ord() function in Python is used to get the Unicode code point of a character. It takes a single character as input and returns an integer representing the Unicode code point of that character.

These functions provide a wide range of string manipulation capabilities in Python. By combining them with slicing, indexing, and the `format()` method, you can work with strings in a variety of ways to suit your programming needs.

Example of using some of string handling functions:

```
my_string = "  Hello, World!  "
print(len(my_string))
```

```
# Output:  18

print(my_string.lower())
# Output:     hello, world!

print(my_string.strip())
# Output:  Hello, World!

print(my_string.startswith("Hello"))
# Output:  True

print(my_string.find("World"))
# Output:  8

print(my_string.replace("World",  "Universe"))
# Output:     Hello, Universe!

print(my_string.split(","))
# Output:  ['   Hello', ' World!   ']

print("-".join(["apple",  "banana",  "cherry"]))
# Output:  apple-banana-cherry

num_str  = "123"
num_int  = int(num_str)
print(num_int)
# Output:  123

num_int  = 123
num_str  = str(num_int)
print(num_str)
# Output:  "123"

character  = 'A'
code = ord(character)
print(code)
# Output:  65

code = 65
character  = chr(code)
print(character)
# Output:  A
```

In the above example, **len(my_string)** returns the length of the string, **my_string.lower()** returns a new string with all characters in lowercase, **my_string.strip()** returns a new string with all leading and trailing whitespace removed, **my_string.startswith("Hello")** returns True because the string starts with "Hello", **my_string.find("World")** returns the index of the first occurrence of "World" in the string, **my_string.replace("World", "Universe")**

returns a new string with all occurrences of "World" replaced with "Universe", **my_string.split(",")** returns a list of substrings separated by commas, and **"-"join(["apple", "banana", "cherry"])** returns a new string that is the concatenation of the strings in the list, separated by a hyphen.

Exercise Problem

1. Write a Python program that takes a user input string and prints the length of the string.
2. Write a Python program that takes a user input string and prints the string in uppercase.
3. Write a Python program that takes a user input string and checks if it is a palindrome (reads the same forwards and backwards).
4. Write a Python program that takes a user input string and counts the number of vowels in the string.
5. Write a Python program that takes a user input string and reverses the order of the words in the string.
6. Write a Python program that takes a user input sentence and converts it to title case (the first letter of each word capitalized).
7. Write a Python program that takes a user input string and removes all the punctuation marks from it.
8. Write a Python program that takes a user input string and counts the occurrences of each character in the string.
9. Write a Python program that takes a user input string and checks if it is an anagram of another user input string.
10. Write a Python program that takes a user input string and replaces all occurrences of a specified word in the string with another word.
11. Write a Python program to reverse a string using slicing.
12. Write a Python function that takes a string as input and returns the number of vowels in the string.
13. Write a Python program to find the first non-repeating character in a string.
14. Write a Python function that takes two strings as input and returns a new string containing only the characters that appear in both strings, in the order they appear in the first string.
15. Write a Python program to check whether a string is a palindrome or not.
16. Data Cleaning: You have a dataset with a column containing messy text data. Write a Python function to clean the text by removing special characters, converting to lowercase, and removing leading/trailing whitespace.
17. Email Validation: Write a Python function to validate an email address. The function should check if the email follows the correct format, such as having the "@" symbol and a valid domain.
18. Password Strength Checker: Write a Python function to check the strength of a password. The function should evaluate the password based on criteria such as length, presence of uppercase letters, lowercase letters, digits, and special characters.
19. Name Formatting: Write a Python function that takes a person's full name as input and returns a formatted version, such as converting the name to title case (capitalizing the first letter of each word) or swapping the first name and last name.
20. Social Media Hashtag Generator: Write a Python function that takes a string as input and generates a hashtag by converting the words to lowercase and adding the "#"

symbol at the beginning. The function should also handle special characters and whitespace.

21. Sentence Analysis: Write a Python program to analyze a sentence. Implement functions to count the number of words, calculate the average word length, identify the longest word, and check if a specific word is present in the sentence.
22. Text Encryption/Decryption: Write a Python program that encrypts or decrypts a given text using a simple encryption algorithm, such as shifting the characters by a fixed number of positions in the ASCII table.