Reading: Format Strings in Python

Estimates effort: 5 mins

Format strings are a way to inject variables into a string in Python. They are used to format strings and produce more human-readable outputs. There are several ways to format strings in Python:

String interpolation (f-strings)

Introduced in Python 3.6, f-strings are a new way to format strings in Python. They are prefixed with 'f' and use curly braces {} to enclose the variables that will be formatted. For example:

```
1. 1
2. 2
3. 3
```

```
1. name = "John"
2. age = 30
3. print(f"My name is {name} and I am {age} years old.")
```

Copied!

This will output:

```
1. 1
```

```
1. My name is John and I am 30 years old.
```

Copied!

str.format()

This is another way to format strings in Python. It uses curly braces {} as placeholders for variables which are passed as arguments in the format() method. For example:

```
1. 1
2. 2
3. 3
```

```
1. name = "John"
2. age = 50
3. print("My name is {} and I am {} years old.".format(name, age))
```

Copied!

This will output:

```
1. 1
```

```
1. My name is John and I am 50 years old.
```

Copied!

% Operator

This is one of the oldest ways to format strings in Python. It uses the % operator to replace variables in the string. For example:

```
1. 1
2. 2
3. 3
```

```
1. name = "Johnathan"
2. age = 30
3. print("My name is %s and I am %d years old." % (name, age))
```

Copied!

This will output:

```
1. 1
```

```
1. My name is Johnathan and I am 30 years old.
```

Copied!

Each of these methods has its own advantages and use cases. However, f-strings are generally considered the most modern and preferred way to format strings in Python due to their readability and performance.

Additional capabilities

F-strings are also able to evaluate expressions inside the curly braces, which can be very handy. For example:

```
1. 1
2. 2
3. 3
```

```
1. x = 10
2. y = 20
3. print(f"The sum of x and y is {x+y}.")
```

Copied!

This will output:

```
1
```

```
The sum of x and y is 30.
```

Copied!

Raw String (r")

In Python, raw strings are a powerful tool for handling textual data, especially when dealing with escape characters. By prefixing a string literal with the letter 'r', Python treats the string as raw, meaning it interprets backslashes as literal characters rather than escape sequences.

Consider the following examples of regular string and raw string:

Regular string:

```
1
2
```

```
regular_string = "C:\new_folder\file.txt"
print("Regular String:", regular_string)
```

Copied!

This will output:

```
1
2
```

```
Regular String:  C:
ew_folderile.txt
```

Copied!

In the regular string regular_string variable, the backslashes (\n) are interpreted as escape sequences. Therefore, \n represents a newline character, which would lead to an incorrect file path representation.

Raw string:

```
1
2
```

```
raw_string = r"C:\new_folder\file.txt"
print("Raw String:", raw_string)
```

Copied!

This will output:

```
1
```

```
Raw String: C:\new_folder\file.txt
```

Copied!

However, in the raw string raw_string, the backslashes are treated as literal characters. This means that \n is not interpreted as a newline character, but rather as two separate characters, '' and 'n'. Consequently, the file path is represented exactly as it appears.

Author: Abhishek Gagneja