Creating the first file

Let's explore how to print multiple lines at once. Let's learn this by printing the "Twinkle, Twinkle, Little Star" poem:

To print multiple lines at once in Python you can use triple-quotes that can be either single (''') or double (""") quotes.

Here's an example:

(In Python)

- print(''' Twinkle, twinkle, little star, How I wonder what you are! Up above the world so high, Like a diamond in the sky. ''')
- print("""Twinkle, twinkle, little star,How I wonder what you are! Up above the world so high, Like a diamond in the sky.""")

Both of these approaches will print the entire poem at once.

Moving on, strings can be enclosed either in single quotes (') or double quotes ("). However, if your string itself contains an apostrophe, it's often more convenient to use double quotes to avoid conflicts.

Let’s understand this with an example:

(In Python)

- # Using single quotes to define a string

  sentence_with_apostrophe = 'It's a beautiful day.'

  # printing the string

  print(sentence_with_apostrophe)

In Python, when you use single quotes to define a string, and if there's an apostrophe within that string, it prematurely ends the string.

If you run this code, you'll encounter an error because the single quote in It prematurely terminates the string. The computer gets confused because it thinks the sentence ends where it sees that apostrophe, and the remaining part (s a beautiful day.') becomes an unexpected piece of code.

Now, let's modify the code to use double quotes to enclose the string:

- # Using double quotes to define a string with an apostrophe

  sentence_with_double_quotes = "It's a beautiful day."

  # Printing the modified string

  print(sentence_with_double_quotes)

  Now, this code will run without any errors and correctly print: "It's a beautiful day."

However, an alternative approach is to use the escape sequence \' to include an apostrophe within a string enclosed in single quotes:

(In python)

- # Using the escape sequence to include an apostrophe

  sentence_with_escape_sequence = 'It\'s a beautiful day.'

  # Printing the string with the escape sequence

  print(sentence_with_escape_sequence)

The escape sequence \' (backslash and a single quote) represents a single quote within the string. The string is still enclosed in single quotes, but the backslash before the apostrophe tells the interpreter that the apostrophe should be treated as part of the string rather than as a string terminator.

When you run this code, it will correctly print: "It's a beautiful day."

So, escape sequences allow you to include special characters, like apostrophes, within strings without causing syntax errors.

Now, let's explore a similar scenario where a string is enclosed within double quotes, and there is a double quote within the string:

Suppose we want to print "He said, "Python is amazing!"".

In this example, the string is initially defined as "He said, "Python is amazing!"". However, including double quotes within the string can lead to potential issues, as Python might interpret the internal double quote as the end of the string. To tackle this, we use the escape sequence \" (backslash and a double quote) within the string like this:

(In Python)

- # Using escape sequence to include a double quote

  sentence_with_double_quotes = "He said, \"Python is amazing!\""

  # Printing the string with the escape sequence

  print(sentence_with_double_quotes)

The escape sequence \" (backslash and a double quote) is utilised within the string to represent a double quote. The backslash signals to the interpreter that the double quote is part of the string content, preventing it from prematurely ending the string. Upon execution, this code will correctly print: "He said, "Python is amazing!""—demonstrating how escape sequences are invaluable for handling special characters within strings, regardless of whether they are enclosed in single or double quotes.

So an escape sequence is like a secret code you tell the computer. It says, "Hey, the character that comes after the backslash is special, so treat it differently – don't take it literally!"

In Python, there are various escape sequences, each serving a specific purpose. Let's explore some of them:

1. **\n - Newline:**

   This is used to break lines. Suppose we want to print “Twinkle Twinkle little star, How I wonder what you are.” But we want “twinkle twinkle little star,” and “how I wonder what you are” in separate lines. Then we will have to use `\n`.

(In Python)

- print("Twinkle twinkle little star,\nHow I wonder what you are.")

(Output: show this output in python)

- Twinkle twinkle little star,

  How I wonder what you are.

1. **\b – Backspace:**

   This is used to erase one character. It is called `backspace` and is equivalent to pressing backspace once. Let’s see an example:

(In Python)

- print("hello \bworld !")

  (Output: show this output in python)

  helloworld !

Even though we put space in between `hello` and `world` the `\b` erased it. And the output we get is without the space.

1. \\ - Backslash:

This is used to insert backward slash in the character string. Let’s see it with an example:

(In Python)

- print("This will insert one \\ (backslash).")

(Output: show this output in python)

- This will insert one \ (backslash).

If you try to print a single backslash (\) without using an escape sequence, it will result in an error.

2. \t – Tab

The escape sequence \t in Python represents a tab character. When used in a string, it tells the computer to insert a horizontal tab space.

Here's an example:

(In Python)

- print("Hello\tWorld")

  The output of this code will be:

  Hello World

In this example, the \t escape sequence inserts a tab space between "Hello" and "World," creating a horizontal gap. This can be useful for formatting text and aligning content in a more structured way.

So these escape sequences help you communicate effectively with the computer, allowing you to include special characters in your strings without causing confusion or errors. They're like little tricks that make sure your instructions are crystal clear.

Now let's imagine we're working with a file path to your favorite pictures. In Python, you might represent the file path as a string. Here's an example:

(In Python)

- # Imagine this is your file path

  favorite_pictures_path = "C:\Users\YourUsername\Pictures\Favorite"

  # Now, let's print the file path

  print("File Path:", favorite_pictures_path)

When you run this code, you might encounter an error. The reason for the error is that backslashes (\) are used as escape characters in Python strings. In the file path, the backslashes are interpreted as escape characters, leading to unexpected behavior.

To address this issue, we can use a raw string, which is denoted by placing an 'r' or 'R' before the string. A raw string treats backslash as literal characters and doesn't interpret them as escape characters. Here's the modified code:

(In Python):

- # Using a raw string for the file path

  favorite_pictures_path = r"C:\Users\YourUsername\Pictures\Favorite"

  # Printing the file path

  print("File Path:", favorite_pictures_path)

  If you run the provided Python code, the output should be:

- File Path: C:\Users\YourUsername\Pictures\Favorite

So the r prefix before the string indicates that it is a raw string, and as a result, the backslashes are treated as literal characters, allowing you to represent Windows file paths without encountering escape character issues.

Now let's understand string formatting!

String Formatting:

Imagine you want to create a message that includes your name. So this is how you will do where we will concatenate a string with a string:

(In Python)

- name = "John"

  # Concatenation approach

  print("My name is " + name)

  The output of the above code will be:

  My name is John

Now let’s use an alternate way: Instead of writing the information separately and combining them, you can use string formatting to make it more straightforward.

(In python)

- name = "Rahul"

  # String formatting approach

  print("My name is {}".format(name))

In the sentence, there's a placeholder {}, which acts like a blank space waiting to be filled.

The format part ensures that the name's value gets filled in that blank space in the sentence.

So, when we print this line, it will say: "My name is John."

Let's add an age variable where a variable is like a named container that can hold and remember different pieces of information for you.

Now, we have two variables: name and age. The modified print statement to include both the name and age in the sentence can be done like this:

(In Python)

- name = "Rahul"

  age = 25

  print("My name is {}. I am {} years old.".format(name, age))

The curly braces {} act as placeholders for name and age values.

The format part fills in the blank spaces with the actual name and age values.

When we print this line, it will say: "My name is Rahul. I am 25 years old."

The order of the variables inside the format method determines the order in which they are inserted into the string. It's important to match the order of the placeholders with the order of the variables to ensure that the values are correctly placed in the resulting string.

Moving on, Python's print() function comes with several parameters that allow you to customize its behavior. Here are some special cases and examples:

(In Python)

- print('Good')

  print('Morning')

  The output of this will be

  Good Morning

So, in the default behavior, the print() function in Python adds a new line character at the end, so each print statement prints its output on a new line.

However, if you want the output of both print statements on the same line, you can achieve this by using the end parameter. Let’s see how:

(In Python)

- # Output in one line

  print('Good', end=' ')

  print('Morning')

  The output will be:

  GoodMorning

By setting the end to a space character (or any other desired separator), the subsequent print statement will be concatenated to the same line instead of starting on a new line.

The end parameter allows for customization of the separator between multiple print statements. Let’s see an example where a space character is used as the separator:

- #Adding a space separator

  print('Good', end=' ')

  print('Morning')

  The output comes out to be

  Good Morning

Where there is a space between Good and Morning.

Now let’s see another example where a comma is used as the separator:

- #Adding a comma separator

  print('Hello', end=',')

  print('World')

  So the output comes out to be

  Hello,World

In addition to the end parameter, Python's print() function also provides the sep parameter, which allows you to customize the separator between multiple arguments within a single print statement. The sep parameter is particularly useful when you want to control the formatting of the output within a single line. Let's take a look at an example:

(In Python)

- # Using the sep parameter to create a custom separator

  print('Apple', 'Banana', 'Orange', sep=' | ')

  The output will be:

  Apple | Banana | Orange

In this example, the print statement with the sep parameter creates a custom separator (' | ') between the specified arguments ('Apple', 'Banana', 'Orange').

Try using a different separator and see how the output changes!

Combining the end and sep parameters in the print() function allows for fine-grained control over both the separator within a single print statement and the behavior at the end of the statement. Let’s see this with an example:

(In Python):

- # Combining end and sep

  ```
  print('Apple', 'Banana', 'Orange', sep=' | ', end=' *** ')

  print('Grapes', 'Melon', 'Kiwi', sep=', ')
  ```

  The Output will be:

  ```
  Apple | Banana | Orange *** Grapes, Melon, Kiwi
  ```

In this example:

The sep parameter is used to set the separator between the words 'Apple', 'Banana', and 'Orange' to ' | '.

The end parameter is used to set the end of the statement to ' *** ' after the first group of fruits.

Another print statement follows, and the sep parameter is again used to set the separator between the words 'Grapes', 'Melon', and 'Kiwi' to ', '.

In summary, the end parameter in the print() function combines the current print statement with the next one, ensuring the output is on a single line. On the other hand, the sep parameter is utilized to specify the separator between multiple values within a single print statement, offering control over the formatting of the output.

Let's consider an example with numerical values using both the sep and end parameters:

-
  ```
  # numerical values, using sep and end

  num1 = 10

  num2 = 20

  num3 = 30

  print("Numbers:", num1, num2, num3, sep=' | ', end=' >>> ')

  print("Sum:", num1 + num2 + num3)
  ```

The sep=' | ' parameter is used to set the separator between the numerical values to a vertical bar followed by a space.

The end=' >>> ' parameter is used to set the end of the print statement to ' >>> '.

Output:

```
Numbers: | 10 | 20 | 30 >>> Sum: 60
```

As a result, the output combines the numerical values with the specified separator and ends with ' >>> ', all in the same line.