**Assignment 6  
  
Q.1. What are escape characters, and how do you use them?**Ans. In Python, escape characters are combinations of a backslash (\) followed by another character. They are used to represent special characters that would otherwise have a different meaning within strings or be difficult to type directly.

Here are some common escape characters and their purposes:

|  |  |  |
| --- | --- | --- |
| Description | Escape Character | Example |
| Newline character, inserts a line break | \n | print("Hello\nWorld") outputs: |
|  |  | Hello |
|  |  | World |
| Horizontal tab character, inserts a tab space | \t | print("This\tis\ta tab") outputs: |
|  |  | This is a tab |
| Double quote character (within double-quoted strings) | \" | print("He said, \"Hello, world!\"") outputs: |
|  |  | He said, "Hello, world!" |
| Single quote character (within single-quoted strings) | \' | print('It\'s a nice day!') outputs: |
|  |  | It's a nice day! |
| Backslash character (to represent a literal backslash) | \\ | print("This is a \\ character") outputs: |
|  |  | This is a \ character |

**Q.2. What do the escape characters n and t stand for?**Ans. In Python, the escape characters \n and \t have specific meanings:

**\n (newline):** This escape character represents a newline character. When included in a string, it inserts a line break, causing the following text to appear on a new line when printed.

Example:  
print("This is on line 1\nThis is on line 2")  
Output :  
This is on line 1  
This is on line 2  
  
**\t (horizontal tab):** This escape character represents a horizontal tab. It inserts a tab space, which is typically a larger space than a single space character. The exact amount of space a tab inserts can vary depending on your terminal or font settings.  
Example:  
print("This\tis\ta tab")  
Output: This is a tab

**3. What is the way to include backslash characters in a string?**  
Ans. There are two main ways to include backslash characters in a Python string:  
**1. Escape Sequence (\\):**  
The most common method is to use an escape sequence, which is a backslash (\) followed by another backslash (\\). This tells Python to interpret the following backslash as a literal backslash character, not as an escape character.  
Here's an example:  
Python Code:  
path = "C:\\Users\\your\_username\\Documents"  
print(path)  
Output:  
C:\Users\your\_username\Documents  
**2. Raw Strings (prefix with r):**  
Another approach is to use raw strings. A raw string is prefixed with the letter r before the opening quotation mark. Raw strings treat all backslashes literally, so you don't need to escape them.  
Here's an equivalent example using a raw string:  
Python Code:  
path = r"C:\Users\your\_username\Documents"  
print(path)  
Output:  
C:\Users\your\_username\Documents

**Q.4. The string "Howl's Moving Castle" is a correct value. Why isn't the single quote character in the word Howl's not escaped a problem?**  
Ans. The single quote character in "Howl's Moving Castle" isn't a problem because the entire string is enclosed in double quotes.  
Explanation:  
Escape characters: These are special combinations with a backslash (\) that tell Python to interpret a character differently. For example, \" is used for a double quote within a double-quoted string.  
The single quote in "Howl's": Since the entire string is wrapped in double quotes, the single quote inside doesn't need to be escaped. It's simply treated as a regular character within the text.  
  
Q.5 How do you write a string of newlines if you don't want to use the n character?  
Ans. There are a couple of ways to write a string of newlines in Python without using the \n character:  
**1. Triple-quoted strings:**Python provides triple-quoted strings (either three single quotes or three double quotes) that can span multiple lines without needing any escape characters. These strings are ideal for situations where you want to include line breaks as part of the string itself, like writing a multi-line poem or block of text.   
Here's an example:  
Python Code:  
multiline\_string = """This is a string

with multiple lines.

You can write as many lines

as you want."""

print(multiline\_string)  
  
Output:  
This is a string  
with multiple lines.  
You can write as many lines  
as you want.  
 **Q.6. What are the values of the given expressions?**

'Hello, world!'[1]

'Hello, world!'[0:5]

'Hello, world!'[:5]

'Hello, world!'[3:]

Ans**. 'Hello, world!'[1]**: This expression accesses the character at index 1 of the string, which is 'e'.

**'Hello, world!'[0:5]**: This expression extracts a substring starting at index 0 (inclusive) and ending before index 5 (exclusive). So, it captures the characters from 'H' to 'o' (excluding the space after 'o'). The value is: 'Hello,'

**'Hello, world!'[:5]**: This is equivalent to the previous expression. An empty starting index defaults to 0, so it starts at the beginning and includes characters up to (but not including) index 5. The value is: 'Hello,'

**'Hello, world!'[3:]:** This expression extracts a substring starting at index 3 (inclusive) and goes all the way to the end of the string. So, it captures everything from 'l' onwards. The value is: 'lo, world!'

**Q.7 What are the values of the following expressions?**

'Hello'.upper()

'Hello'.upper().isupper()

'Hello'.upper().lower()

Ans.   
**'Hello'.upper():**  
The .upper() method converts all lowercase characters in the string 'Hello' to uppercase.  
Value: 'HELLO'

**'Hello'.upper().isupper():**

This combines two methods:  
.upper() from the previous step is chained, so it's applied to the already uppercase string 'HELLO'. Since 'HELLO' only contains uppercase characters, there's no further change.  
.isupper() checks if all characters in the string are uppercase.  
Value: True (because 'HELLO' only has uppercase letters)

**'Hello'.upper().lower():**  
Similar to the previous expression, .upper() is applied first, resulting in 'HELLO'.  
Then, .lower() is applied, converting all characters in 'HELLO' to lowercase.  
Value: 'hello'  
  
**Q.8 What are the values of the following expressions?**

'Remember, remember, the fifth of July.'.split()

'-'.join('There can only one.'.split())  
Ans.

1. **'Remember, remember, the fifth of July.'.split()**

This expression uses the .split() method on the string "Remember, remember, the fifth of July.". The .split() method splits the string into a list of words, using whitespace (spaces, tabs, newlines) as the default delimiter (separator) between words. The split string includes the commas (,) and the period (.) as separate elements in the list because they are considered delimiters by default.  
Output : ['Remember,', 'remember,', 'the', 'fifth', 'of', 'July.']

1. **'-'.join('There can only one.'.split())**  
   This expression involves two methods chained together:  
   .split() splits the string "There can only one." into a list of words based on whitespace.

.join() joins the elements of the resulting list using the specified separator ('-' in this case).  
Output : 'There-can-only-one.'

**Q.9. What are the methods for right-justifying, left-justifying, and centering a string?**

Ans. There are two common approaches for right-justifying, left-justifying, and centering strings in Python:  
**1. String Methods (ljust(), rjust(), center()):**  
Python provides built-in methods for each type of justification:  
.ljust(width, fillchar): Left-justifies the string within a specified width, padding the right side with the fillchar (default is space).

.rjust(width, fillchar): Right-justifies the string within a specified width, padding the left side with the fillchar.

.center(width, fillchar): Centers the string within a specified width, padding both sides with the fillchar.

Python Code:  
text = "Hello, world!"

width = 20  
print("Left-justified:", text.ljust(width))

print("Right-justified:", text.rjust(width))

print("Centered:", text.center(width))  
  
Output:  
Left-justified: Hello, world!

Right-justified: Hello, world!

Centered: Hello, world!   
  
**2. f-strings (formatted string literals):**  
  
f-strings (introduced in Python 3.6) offer a more concise way to format strings with alignment. You can use format specifiers within curly braces ({}) to control the justification:

{: >width}: Right-justifies the string within the specified width.

{: <width}: Left-justifies the string within the specified width.

{: ^width}: Centers the string within the specified width.

Here's the equivalent example using f-strings:  
Python Code:  
text = "Hello, world!"

width = 20

print(f"Left-justified: {text:<{width}}")  
print(f"Right-justified: {text:>{width}}")  
print(f"Centered: {text: ^{width}}")  
Output:

Left-justified: Hello, world!

Right-justified: Hello, world!

Centered: Hello, world!   
  
**10. What is the best way to remove whitespace characters from the start or end?**Ans. **1. Removing all Whitespace:**If you want to remove all whitespace characters (spaces, tabs, newlines, etc.) from a string, the translate() method with a translation table is a good choice:  
Python Code:  
import string  
def remove\_all\_whitespace(text):  
"""Removes all whitespace characters from a string.  
Args:  
text: The string to remove whitespace from.  
Returns:  
The string with all whitespace characters removed.  
“"”  
remove\_chars = string.whitespace # Predefined string containing whitespace characters  
translation\_table = str.maketrans('', '', remove\_chars)  
return text.translate(translation\_table)  
text = " This string has whitespace. "  
text\_without\_whitespace = remove\_all\_whitespace(text)  
print(text\_without\_whitespace)   
Output: Thisstringhaswhitespace.  
  
**2. Removing Leading/Trailing Whitespace:**  
For removing only leading and/or trailing whitespace, Python offers three dedicated methods:  
strip(): The most versatile option, it removes both leading and trailing whitespace characters by default.  
lstrip(): Removes leading whitespace characters only.  
rstrip(): Removes trailing whitespace characters only.

Python Code:  
text = " Hello, world! "

text\_stripped = text.strip() # Removes both leading and trailing whitespace

text\_lstripped = text.lstrip() # Removes leading whitespace only

text\_rstriped = text.rstrip() # Removes trailing whitespace only

print("Original:", text)

print("Stripped:", text\_stripped)

print("Left-stripped:", text\_lstripped)

print("Right-stripped:", text\_rstriped)  
  
Output: Original:   
Hello, world!   
Stripped: Hello, world!  
Left-stripped: Hello, world!   
Right-stripped: Hello, world