

Contents

1. Keywords	2
2. Basic Data Types	2
3. Variable and Input	5
4. Logic and Comparison Operators	12
5. Conditional Statement	14
6. Loops	19
7. Break and Continue	23
8. Functions	24
9. Recursive Functions in Python	28
Lab Questions	29

1. Keywords

Keywords are predefined, reserved words used in programming that have their special meanings. Keywords are part of a syntax. They are reserved for specific purpose in the language. Currently there are 36 keywords in python. Python Keywords cannot be used as identifiers (name given to a variable or a function). If we use keyword as an identifier then python will throw syntax error. Here, is a list of some python keywords.

Keywords	Remarks
if, elif, else, for, while, break, continue, pass	Used in control flow structures
def, class	Used in function and class definition
try, except, finally, raise	Used in exception handling
global, nonlocal	Used for namespace and scope
in	Used in Iterators
True, False, and, or, not	Used for Boolean and logical operations
del, return, as	Used in object creation and management
import, from	Used in import and module Management

2. Basic Data Types

Data types are the fundamental building blocks of programming language. Data type is a classification that specifies which type of value a variable can hold and what type of operation can be performed on the variable. Python has a dynamic type casting, which means that we don't need to explicitly specify the data type of a variable when we declare it. Python infers the data type based on the value assigned to the variable. Python data types are classes and variables are the object of these classes.

Basic data types in python are:

1. **Integers:** They are represented by int class. It contains positive and negative whole numbers without fractions. In python there is no limit to how long an integer value can be. In python a non-decimal integers like binary, octal and hexadecimal can be represented by adding a prefix 0B or 0b, 0O or 0o, 0X or 0x respectively.

```

Microsoft Windows [Version 10.0.22621.3447]
(c) Microsoft Corporation. All rights reserved.

C:\Users\user>python
Python 3.9.7 (tags/v3.9.7:1016ef3, Aug 30 2021, 20:19:38) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> type(105)
<class 'int'>
>>> 0b1010
10
>>> type(0b1010)
<class 'int'>
>>> 0xff
255

```

**Note: type() is a function to determine the type of data type in python*

Here, we have used a python interpreter in command on windows operating system. When we use type() to determine the type of a number 105 the interpreter prints the type as int. Everything in python is class so the interpreter prints the output as class 'int' since the value belong to integer class. 0b represents a binary number. So now 1010 is a binary number and the interpreter outputs the decimal value for 1010 binary number. Similarly, 0xff represent a hexadecimal number ff and the interpreter outputs the corresponding decimal value.

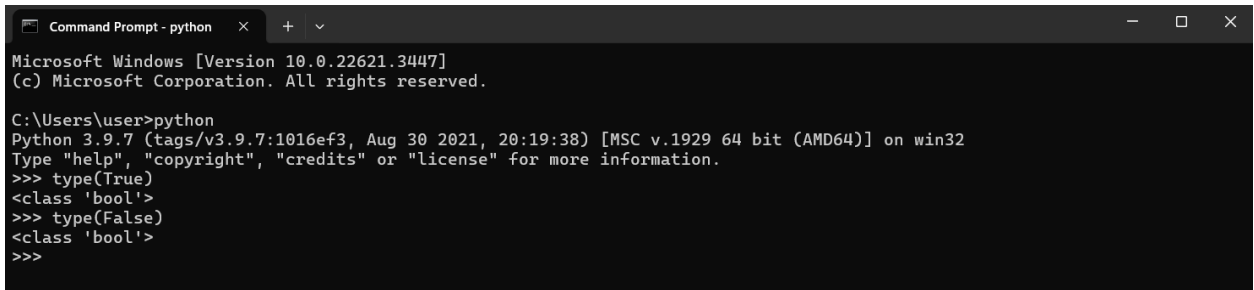
2. Floating-point: Floating point values are represented by float class. It contains real numbers with decimal point. Optionally, e or E is added at the end of a floating-point number to specify the scientific notation.

```

C:\Users\user>python
Python 3.9.7 (tags/v3.9.7:1016ef3, Aug 30 2021, 20:19:38) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> type(10.55)
<class 'float'>
>>> 105e5
10500000.0
>>> 105e-5
0.00105
>>> |

```

3. Boolean: Boolean data types are represented by bool class. They have two values true and false. Non- Boolean objects can be evaluated in the Boolean context as well and determined to be true or false.



```

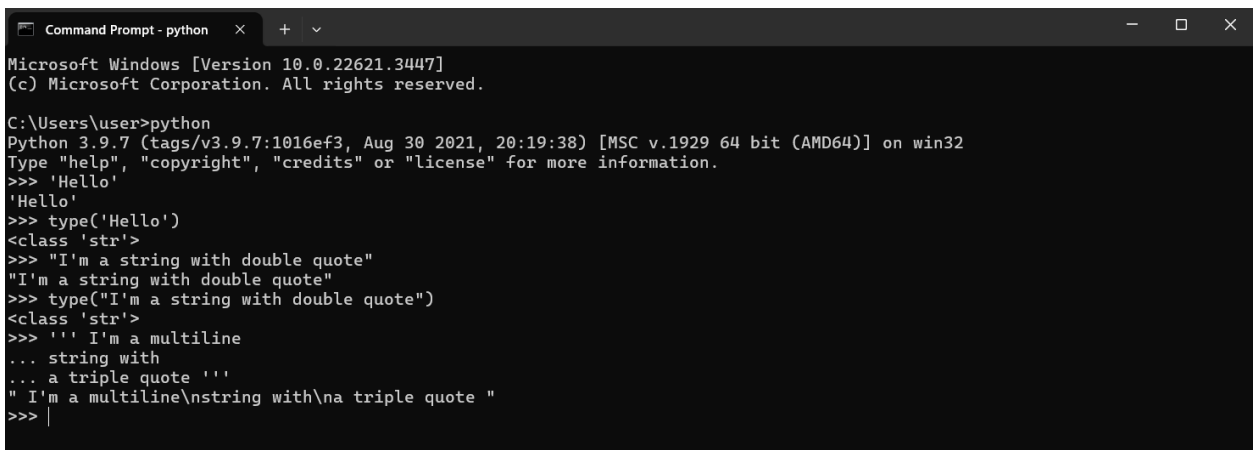
Microsoft Windows [Version 10.0.22621.3447]
(c) Microsoft Corporation. All rights reserved.

C:\Users\user>python
Python 3.9.7 (tags/v3.9.7:1016ef3, Aug 30 2021, 20:19:38) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
>>>

```

**Note: True has 'T' capital. Interpreter throws an error if true or false is used.*

4. String: A string data type is represented by str class. String is a collection of one or more character put in a single quote ('Hello'), double quote ("I'm a string with double quote") and triple quote ("I'm a multiline string with a triple quote").



```

Microsoft Windows [Version 10.0.22621.3447]
(c) Microsoft Corporation. All rights reserved.

C:\Users\user>python
Python 3.9.7 (tags/v3.9.7:1016ef3, Aug 30 2021, 20:19:38) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 'Hello'
'Hello'
>>> type('Hello')
<class 'str'>
>>> "I'm a string with double quote"
"I'm a string with double quote"
>>> type("I'm a string with double quote")
<class 'str'>
>>> ''' I'm a multiline
... string with
... a triple quote '''
"I'm a multiline\nstring with\na triple quote "
>>> |

```

Here, A string 'Hello' was created using single quote. "I'm a string with double quote" is a string created using double quote, the string begins with " and ends with " and everything inside the double quote is string data. "I'm a multiline string with a triple quote" is created using triple quote, the string begins with " and ends with ". Triple quote can be used to create a multi-line string as shown in the above terminal.

**note "I'm a multiline\nstring with\na triple quote " is also a valid string as we can use "I'm a multiline\nstring with\na triple quote " to create a string.*

5. None: It is represented by NoneType object. None is a special constant representing the absence of a value or a null value.

```
Command Prompt - python
C:\Users\user>python
Python 3.9.7 (tags/v3.9.7:1016ef3, Aug 30 2021, 20:19:38) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> type(None)
<class 'NoneType'>
>>>
```

**Note: None has 'N' capital. Interpreter throws an error if none is used.*

3. Variable and Input

Variables are used to store data such as integers, floats, strings in python. In python there is no need of declaring the variables. Variable are created when first value is assigned to it. Python is dynamically typed i.e. type is inferred based on the value assigned to it.

Rules for naming a variable in Python:

- Variable name can contain letters, digits and underscore (_).
- Variable name cannot start with digit.
- Variable name cannot be keyword.

```
variables_1.py
1 x = 10
2 y = 10.55
3 z = "Hello"
4 print(x)
5 print(y)
6 print(z)
7 print(type(x))
8 print(type(y))
9 print(type(z))

10
10.55
Hello
<class 'int'>
<class 'float'>
<class 'str'>
PS E:\python programming>
```

Here, x holds an integer value 10, y holds a float value 10.55, and z holds a string value "Hello".

x is integer variable; y is float variable and z is a string variable. We now can perform respective operations based on the datatype of the variable.

In python we can take user input using input() function. This function reads a line from a keyboard and it returns a string.



```
input.py > ...  
1 name = input("Enter your name")  
2 print(name)  
  
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS  
  
PS E:\python_programming> python -u "e:\python_programming\input.py"  
Enter your namepravin  
pravin
```

If we want to input to be of other type then we have to convert it accordingly. If we want to convert it to integer then we can do it using int() method. Here, int() is a function that is used to convert a value to an integer.

```
5 age = int(input("Enter your are:"))
6 print(age)
7 print(type(age))
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS E:\python_programming> python -u "e:\python_programming\tempCodeRunnerFile.py"
Enter your are:10
10
<class 'int'>
PS E:\python_programming> 
```

Basic Arithmetic operators in python:

- Addition (+), Subtraction (-), Multiplication (*), Division (/), Floor Division (//), Modulus (%), Exponentiation (**)

```
arithmeticop.py > ...
1 a = 10
2 b = 3
3
4 # Perform arithmetic operations
5 addition_result = a + b
6 subtraction_result = a - b
7 multiplication_result = a * b
8 division_result = a / b
9 floor_division_result = a // b
10 modulus_result = a % b
11 exponentiation_result = a ** b
12
```

```

arithmicop.py > ...
12
13 # Print results
14 print("Addition:", addition_result)
15 print("Subtraction:", subtraction_result)
16 print("Multiplication:", multiplication_result)
17 print("Division:", division_result)
18 print("Floor Division:", floor_division_result)
19 print("Modulus:", modulus_result)
20 print("Exponentiation:", exponentiation_result)

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

PS E:\python_programming> python -u "e:\python_programming\arithmicop.py"
Addition: 13
Subtraction: 7
Multiplication: 30
Division: 3.3333333333333335
Floor Division: 3
Modulus: 1
Exponentiation: 1000

```

Working with strings in python:

```

strings.py > ...
1 str1 = "Hello"
2 str2 = "World"
3 result1 = str1 + " " + str2           #concatination of String
4 print("After concatination: "+result1)
5 result2 = result1 * 3                 #Repetition of String
6 print("After Repetition: "+result2)
7 first_char = str1[0]                 #indexing a String
8 last_char = str1[-1]
9 print("First char: "+ first_char)
10 print("Last char: "+ last_char)
11 first_word = result1[:5]             #slicing a String
12 last_word = result1[6:]
13 print("First word: "+ first_word)
14 print("last word: "+ last_word)

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

> python -u "e:\python_programming\strings.py"
After concatination: Hello World
After Repetition: Hello WorldHello WorldHello World
First char: H
Last char: o
First word: Hello
last word: World
PS E:\python_programming>

```


The provided code demonstrates several fundamental string operations in Python. Initially, it concatenates two strings, `str1` ("Hello") and `str2` ("World"), with a space in between, resulting in `result1` ("Hello World"). It then repeats this concatenated string three times to form `result2` ("Hello WorldHello WorldHello World").

Indexing allows you to access individual characters in a string based on their position. Python uses zero-based indexing, meaning the first character of a string is at index 0. `str1[0]`: Accesses the first character of `str1`, which is "H". `str1[-1]`: Uses negative indexing to access the last character of `str1`, which is "o". Negative indexing counts from the end of the string, with -1 being the last character, -2 being the second last, and so on. Slicing allows you to extract a substring from a string by specifying a start and end index. The syntax is `string[start:end]`, where the start index is inclusive and the end index is exclusive.

`result1[:5]` : Slices the string from the beginning up to, but not including, index 5. This extracts the first five characters of `result1`, which are "Hello".

`result1[6:]` : Slices the string from index 6 to the end of the string. This extracts the substring starting from the 7th character to the end, which is "World".

Working with string functions:



```
string_function.py > ...
1 str1 = "Hello"
2 str2 = "WORLD"
3 print(str1.upper())
4 print(str1.lower())
5 str3 = "Hello,namaste,nepal"
6 print(str3.split(","))
7 str4 = "  Hello  "
8 print(str4.strip())
9 str5 = "Hello Nepol"
10 print(str5.replace("Nepol","Nepal"))
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS E:\python_programming> python -u "e:\python_programming\string_function.py"
HELLO
hello
['Hello', 'namaste', 'nepal']
Hello
Hello Nepal
PS E:\python_programming>
```

The `upper()` method converts all characters in a string to uppercase, while `lower()` does the opposite by converting all characters to lowercase. The `split()` method splits a string into a list of substrings

based on a specified separator (in this case, ,). `strip()` removes leading and trailing whitespace characters from a string. Finally, `replace()` replaces occurrences of a specified substring within a string with another substring.

For instance, `str1.upper()` converts the string "Hello" to uppercase, yielding "HELLO", while `str1.lower()` does the opposite, resulting in "hello". `str3.split(",")` splits the string "Hello,namaste,nepal" into a list of substrings based on the comma separator, resulting in ["Hello", "namaste", "nepal"]. `str4.strip()` removes leading and trailing whitespace characters from the string " Hello ", resulting in "Hello". Lastly, `str5.replace("Nepol", "Nepal")` replaces the substring "Nepol" with "Nepal" in the string "Hello Nepol", producing "Hello Nepal".

String Formatting:

We will discuss two types of formatting.

- a. `format()` method
- b. F-Strings (Python 3.6+)



```
stringformatting.py > ...
1 name = "Ram Thapa"
2 age = 25
3 result = "My name is {} and age is {}".format(name,age) #format() method
4 print(result)
5 result = f"My name is {name} and age is {age}"           #F' strings
6 print(result)

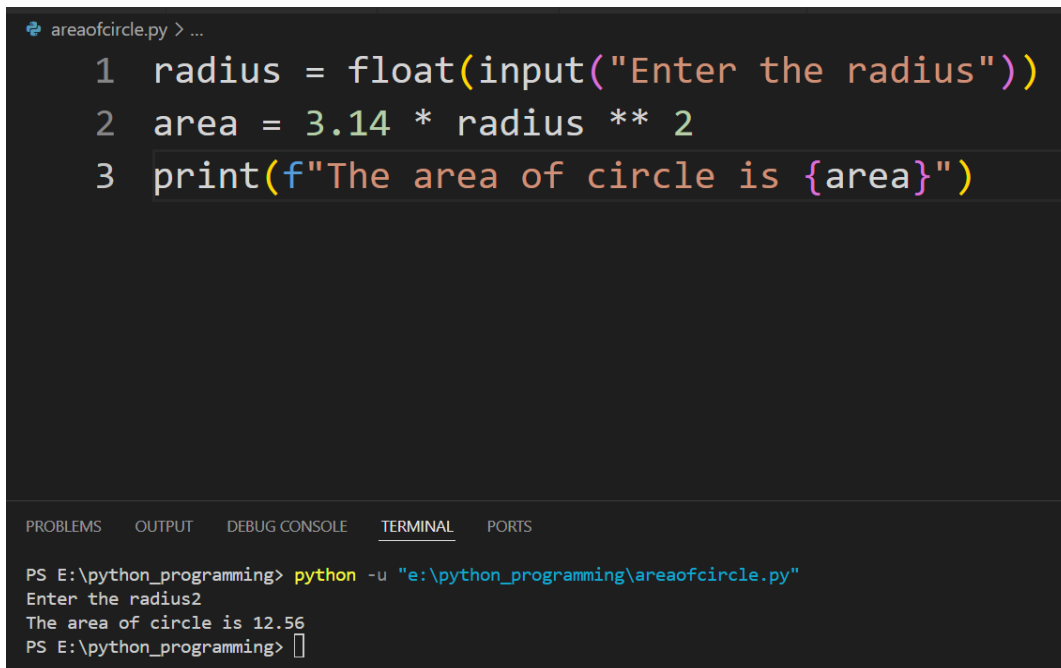
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS E:\python_programming> python -u "e:\python_programming\stringformatting.py"
My name is Ram Thapa and age is 25
My name is Ram Thapa and age is 25
PS E:\python_programming>
```

The first method discussed is the `format()` method. With this approach, you construct a string template containing placeholders (`{}`) to indicate where variables or expressions should be inserted. Then, you call the `format()` method on the template string, passing the variables or expressions as arguments. Inside the placeholders, you can specify the order of variables or use numbered placeholders for explicit positioning.

f-strings provide a more concise and intuitive way to format strings by allowing you to directly embed variables and expressions within curly braces `{ }` within the string literal. This is achieved by prefixing the string literal with an `'f'` or `'F'`. With f-strings, there's no need to explicitly call a method like `format()`; instead, the variables or expressions within the curly braces are automatically evaluated and replaced with their values at runtime. This results in more readable and maintainable code, especially when dealing with complex string formatting scenarios.

In the first example, `result = "My name is { } and age is { }".format(name, age)`, the `format()` method replaces the curly braces `{ }` in the template string with the values of `name` and `age`, resulting in `"My name is Ram Thapa and age is 25"`. In the second example, `result = f"My name is {name} and age is {age}"`, f-strings allow you to directly embed variables and expressions within curly braces `{ }` within the string literal, making the code more concise and readable. Both techniques offer powerful ways to format strings in Python, catering to different preferences and requirements.

Q. WAP in python to input the radius of circle and output “The area of circle is”



```
areaofcircle.py > ...
1 radius = float(input("Enter the radius"))
2 area = 3.14 * radius ** 2
3 print(f"The area of circle is {area}")

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS E:\python_programming> python -u "e:\python_programming\areaofcircle.py"
Enter the radius2
The area of circle is 12.56
PS E:\python_programming> 
```

It first prompts the user to input the radius of the circle using the `input()` function, which returns a string. The `float()` function is then used to convert the input string to a floating-point number, ensuring that the radius can be a decimal value. Next, it calculates the area of the circle using the formula `area = 3.14 * radius ** 2`, where `radius` is the user-provided value. Finally, the `print()` function displays the result using an f-string, where the curly braces `{ }` are replaced with the value

of the area variable, resulting in a message like "The area of the circle is 78.5" if the radius input by the user was 5. This code effectively demonstrates how to receive user input, perform a calculation, and display the result in a formatted message in Python.

4. Logic and Comparison Operators

Logic operators evaluate logical expressions and return Boolean values (True or False), while comparison operators compare two values and return a Boolean result based on the comparison.

- Logical Operators (and , or, not)
- Comparison Operators (==, !=, <, >, <=, >=)

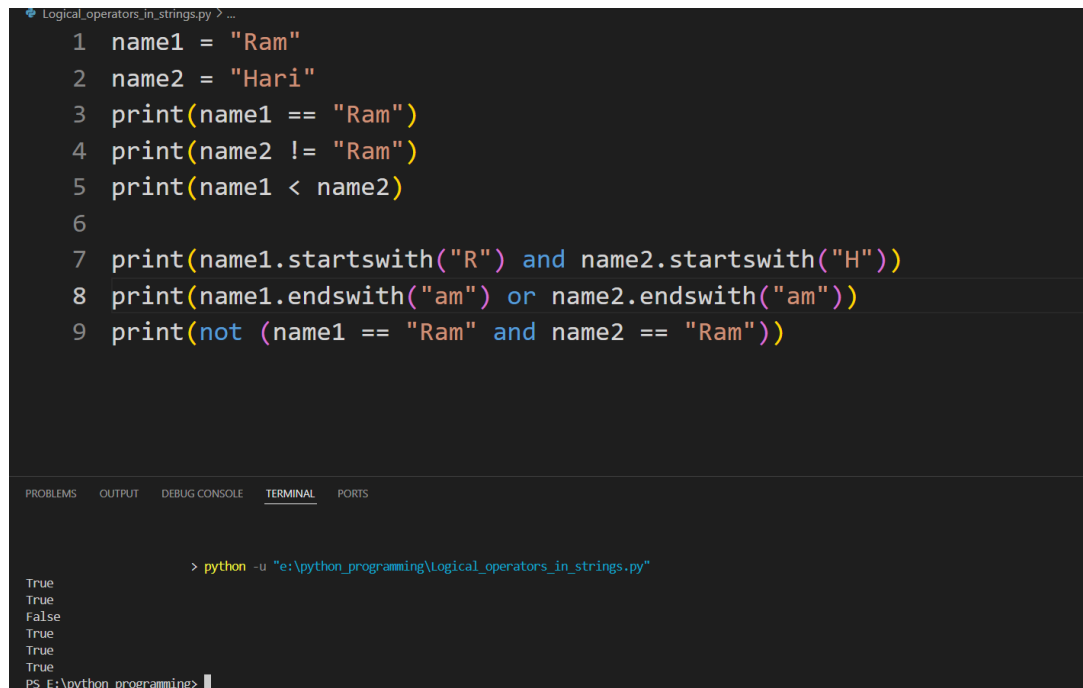
```
logical_and_comparision.py > ...
1 x = 5
2 y = 10
3 # Logic operators
4 print(x < 10 and y > 5)
5 print(x < 10 or y < 5)
6 print(not(x < 10 and y > 5))
7 # Comparison operators
8 print(x == 5)
9 print(y != 5)
10 print(x < y)
11 print(y >= 10)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
> python -u "e:\python_programming\logical_and_comparision.py"
True
True
False
True
True
True
True
True
PS E:\python_programming> 
```

Here, several logic and comparison operators are applied to variables x and y. For logic operators, the and operator evaluates to True for the first print statement because both conditions, x < 10 (True) and y > 5 (True), are satisfied. Similarly, the or operator in the second print statement evaluates to True because at least one condition is True (x < 10). The not operator in the third print

statement negates the result of the condition $x < 10$ and $y > 5$, which is True, resulting in False. Regarding comparison operators, $x == 5$ is True because x is equal to 5, $y != 5$ is True because y is not equal to 5, $x < y$ is True because 5 is less than 10, and $y >= 10$ is True because 10 is greater than or equal to 10. These results demonstrate how logic and comparison operators function in Python, enabling developers to create conditional expressions and make decisions based on the evaluated conditions.



```
Logical_operators_in_strings.py > ...
1 name1 = "Ram"
2 name2 = "Hari"
3 print(name1 == "Ram")
4 print(name2 != "Ram")
5 print(name1 < name2)
6
7 print(name1.startswith("R") and name2.startswith("H"))
8 print(name1.endswith("am") or name2.endswith("am"))
9 print(not (name1 == "Ram" and name2 == "Ram"))

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

> python -u "e:\python_programming\Logical_operators_in_strings.py"
True
True
False
True
True
True
True
PS E:\python_programming>
```

In this example, the code compares two string variables `name1` and `name2`, which contain "Ram" and "Hari" respectively. The comparison operators check if `name1` is equal to "Ram" (`name1 == "Ram"`) and if `name2` is not equal to "Ram" (`name2 != "Ram"`). Additionally, the less than operator (`name1 < name2`) compares the strings lexicographically, resulting in True because "Ram" comes before "Hari" alphabetically. For logic operators, `startswith()` verifies if both names start with the respective letters "R" and "H", returning True. `endswith()` checks if at least one name ends with the substring "am", yielding True for `name1`. Finally, the `not` operator negates the expression `name1 == "Ram" and name2 == "Ram"`, indicating True because both names are not equal to "Ram".

5. Conditional Statement

Conditional statements are fundamental for controlling the flow of a program and implementing decision-making logic. Python supports several types of conditional statements:

- **if Statement:** The if statement evaluates a condition and executes the block of code within it if the condition is true.
- **if-else Statement:** The if-else statement provides an alternative block of code to execute if the condition in the if statement is false.
- **if-elif-else Statement:** The if-elif-else statement is used to check multiple conditions. The first block of code with a true condition is executed, and the rest are skipped.

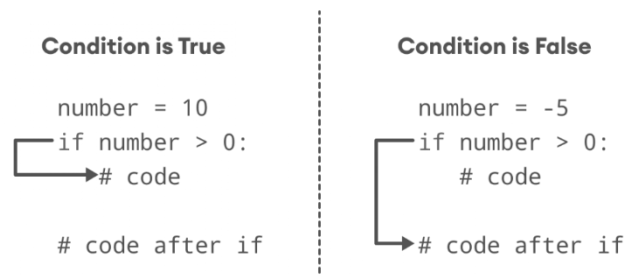
if Statement

Syntax:

```
if condition:  
    # body of if statement
```

Here, if the condition of the if statement is:

- **True** - the body of the if statement executes.
- **False** - the body of the if statement is skipped from execution.



Indentation: Python relies on indentation (whitespace at the beginning of a line) to define scope in the code.

Here is a basic if statement in Python. It starts by assigning the value 10 to the variable x. The if statement then checks if x is greater than 5 ($x > 5$). Since this condition is true, the indented block of code within the if statement is executed.

```
if_statement.py > ...
1 x = 10
2 if x > 5:
3     print("x is greater than 5")

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS E:\python_programming> python -u "e:\python_programming\if_statement.py"
x is greater than 5
PS E:\python_programming> |
```

In Python, the code block inside an if statement must be indented. The lack of indentation here will result in an `IndentationError`.

```
if_statement.py > ...
1 x = 10
2 if x > 5:
3 print("x is greater than 5")

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS E:\python_programming> python -u "e:\python_programming\if_statement.py"
File "e:\python_programming\if_statement.py", line 3
print("x is greater than 5")
^
IndentationError: expected an indented block
PS E:\python_programming> |
```

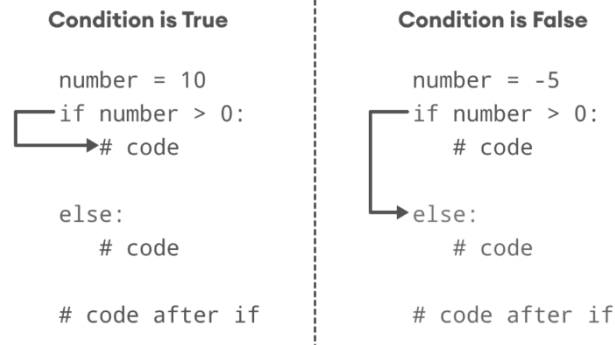
if-else Statement:

Syntax:

```
if condition:
    # body of if statement
else:
    # body of else statement
```

Here, if the condition inside the if statement evaluates to

- True - the body of if executes, and the body of else is skipped.
- False - the body of else executes, and the body of if is skipped



```
if_else.py > ...
1 x = 10
2 if x > 15:
3     print("x is greater than 15")
4 else:
5     print("x is not greater than 15")

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

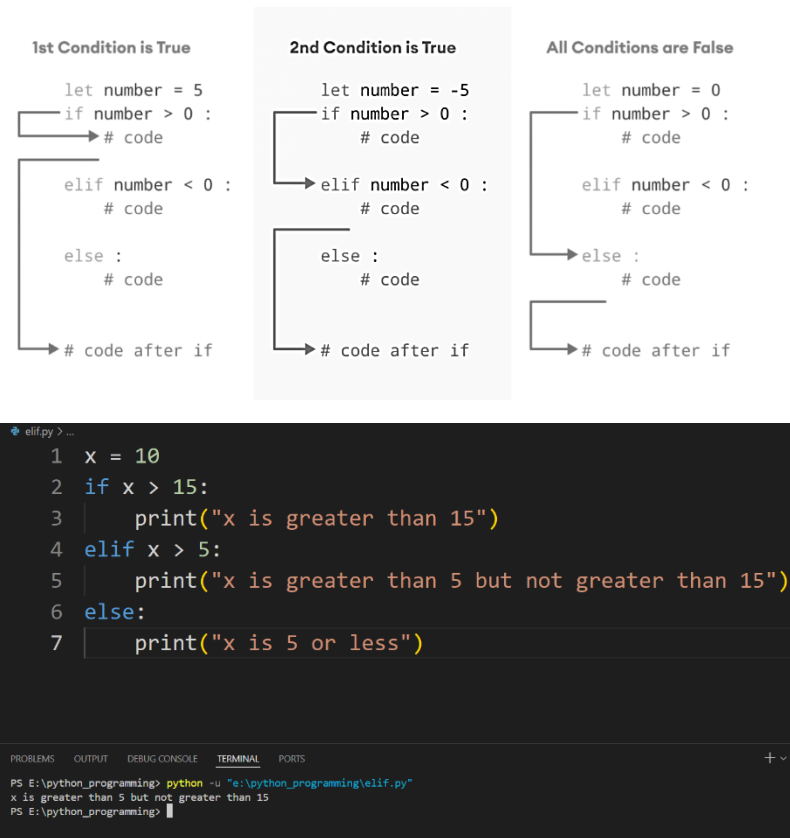
PS E:\python_programming> python -u "e:\python_programming\if_else.py"
x is not greater than 15
PS E:\python_programming> []
```

Here, if-else statement to check whether the variable `x` is greater than 15. Initially, the variable `x` is assigned the value 10. The if statement evaluates the condition `x > 15`. Since the condition is false, so the code inside the if block (`print("x is greater than 15")`) is skipped. Instead, the else block is executed, printing "x is not greater than 15" to the console.

if-elif-else Statement:

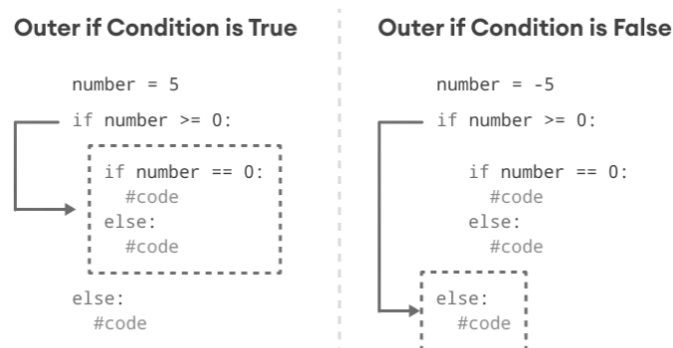
Syntax:

```
if condition1:
    # code block 1
elif condition2:
    # code block 2
else:
    # code block 3
```

Here the code is using an if-elif-else statement to evaluate multiple conditions for the variable x, which is assigned the value 10. The if statement first checks if x is greater than 15. Since this condition is false, so the code moves to the elif statement. The elif statement checks if x is greater than 5 this condition is true, and the code inside the elif block (print("x is greater than 5 but not greater than 15")) is executed. The else block is ignored because an earlier condition was met.

Nested if Statements:



It is possible to use conditional statements inside one another to check multiple conditions at different levels. This is known as nested statements.

```
nested_if.py > ...
1 x = 10
2 y = 20
3 if x > 5:
4     if y > 15:
5         print("x is greater than 5 and y is greater than 15")
6     else:
7         print("x is greater than 5 but y is not greater than 15")
8 else:
9     print("x is 5 or less")

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS E:\python_programming> python -u "e:\python_programming\nested_if.py"
x is greater than 5 and y is greater than 15
PS E:\python_programming> |
```

Here, nested if statements are used to evaluate conditions involving two variables, x and y, with values 10 and 20, respectively. The outer if statement first checks if x is greater than 5. Since the condition is true, and the code proceeds to the inner if statement. The inner if statement then checks if y is greater than 15. Given that this condition is also true, so the code inside the inner if block (`print("x is greater than 5 and y is greater than 15")`) is executed.

if Shorthand, Ternary Operator if...else

Ternary operator:

Syntax: true_value if condition else false_value

```
shorthand_if.py > ...
1 # short hand if
2 # the if statement can be simplified into one line
3 number = 10
4 if number>0: print('Positive')
5
6 #ternary operator if .. else
7 percentage = 80
8 result = "Distinction" if percentage >= 80 else "Not Distinction"
9 print(result)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS E:\python_programming> python -u "e:\python_programming\shorthand_if.py"
Positive
Distinction
PS E:\python_programming> |
```

Here, a shorthand if statement and a ternary operator are demonstrated. First, the shorthand if statement evaluates if the variable number (which is 10) is greater than 0. Since the condition is true, it prints "Positive" in a single line. Next, the ternary operator checks if the variable percentage (which is 80) is greater than or equal to 80. Since this condition is true, the variable result is assigned the string "Distinction".

6. Loops

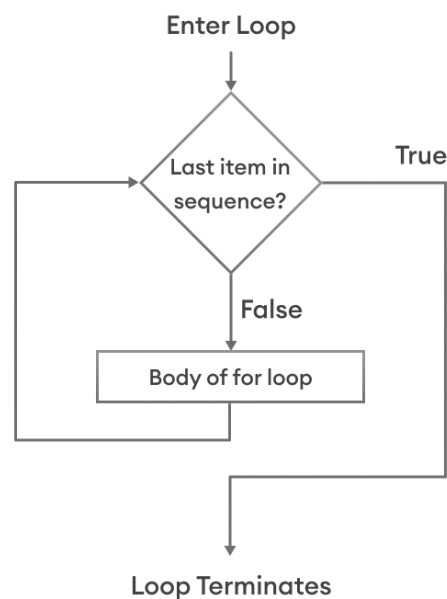
Python supports two main types of loops.

- for loop: for loops iterate over a sequence (such as a list, tuple, string, or range) and execute a block of code for each item in the sequence.
- while loop: while loops continue to execute a block of code as long as a specified condition is true.

Flowchart of Python for Loop:

Syntax:

```
for val in sequence:  
    # statement(s)
```



For loop with range ():

```
for_range.py > ...
1 for i in range(5):
2     print(i)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS E:\python_programming> python -u "e:\python_programming\for_range.py"
0
1
2
3
4
PS E:\python_programming> 
```

In this example, a for loop is used with the range function to print numbers from 0 to 4. The range(5) function generates a sequence of numbers starting from 0 up to, but not including, 5. The for loop iterates over each number in this sequence, and during each iteration, the current number (i) is printed. This results in the output of numbers 0, 1, 2, 3, and 4.

Some other forms of range():

- range(2, 6): Output: 2, 3, 4, 5
- range(1, 10, 2): Output: 1, 3, 5, 7, 9
- range(10, 0, -2): Output: 10, 8, 6, 4, 2
- The range function is versatile and can be used with one, two, or three arguments to generate sequences of numbers with specific start, stop, and step values. The step value can be positive or negative, allowing for both ascending and descending sequences.

For loop with string :

```
for_string.py > ...
1 str = "Ram Thapa"
2 for ch in str:
3     print(ch)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS E:\python_programming> python -u "e:\python_programming\for_string.py"
R
a
m

T
h
a
p
a
```

Here, the code defines a string str with the value "Ram Thapa" and uses a for loop to iterate over each character in the string. The loop variable ch takes on the value of each character in the string, one at a time. Inside the loop, the print(ch) statement outputs the current character to the console.

As a result, each character in "Ram Thapa" is printed on a new line, producing the following output: R, a, m, (a space), T, h, a, p, a.

Q. WAP to check whether a number is even or odd within a range input by the user.

```
odd_even_in_a_range.py > ...
1 start = int(input("Enter the starting value"))
2 end = int(input("Enter the ending value"))
3 for num in range(start, end + 1):
4     if num % 2 == 0:
5         print(f"{num} is even")
6     else:
7         print(f"{num} is odd")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Enter the starting value1
Enter the ending value10
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
10 is even
```

Q. WAP to check whether a character in a string is vowel or not. The string must be input by the user.

```
vowel_or_not_in_c.py > ...
1 vowel = "aeiouAEIOU"
2 word = input("Enter a word")
3 for ch in word:
4     if ch in vowel:
5         print(f"{ch} is a vowel")
6     else:
7         print(f"{ch} is a consonant")
```

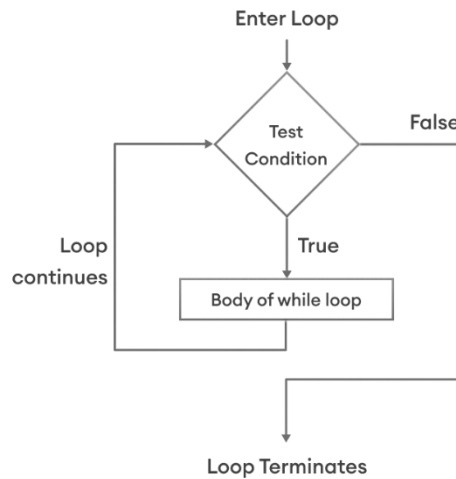
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS E:\python_programming> python -u "e:\python_programming\vowel_or_not_in_c.py"
Enter a wordNepal
N is a consonant
e is a vowel
p is a consonant
a is a vowel
l is a consonant
```

Flowchart of Python while Loop:

Syntax:

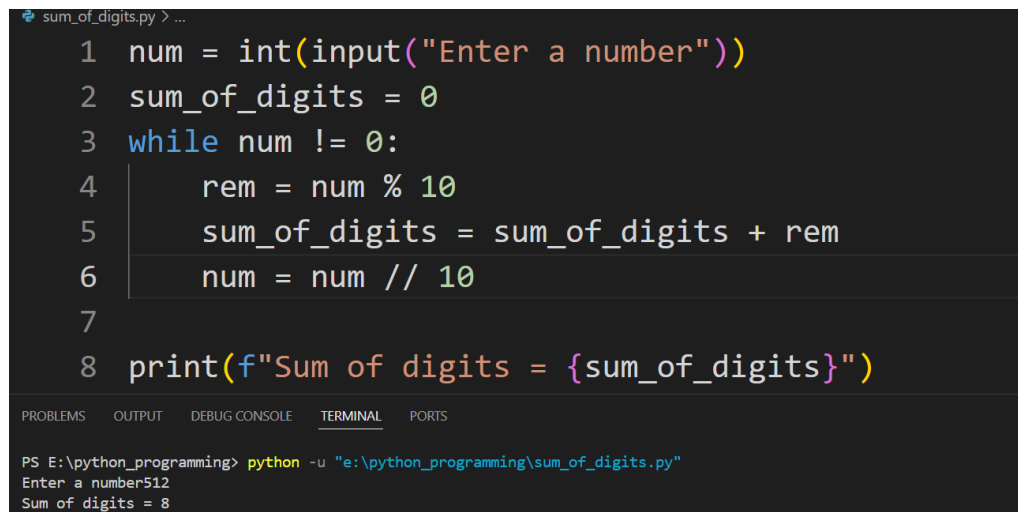
```
while condition:  
    # body of while loop
```



```
while.py > ...  
1 num = 1  
2 while num<=5:  
3     print(num)  
4     num = num + 1  
  
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS  
PS E:\python_programming> python -u "e:\python_programming\while.py"  
1  
2  
3  
4  
5
```

Here, the code initializes a variable num to 1 and then enters a while loop that continues as long as num is less than or equal to 5. Inside the loop, it prints the current value of num and then increments num by 1. This process repeats, with num being printed and then increased each time, until num exceeds 5, at which point the loop terminates. Consequently, the output is the sequence of numbers from 1 to 5, each printed on a new line.

Q. WAP to calculate the sum of digits of a number.



```

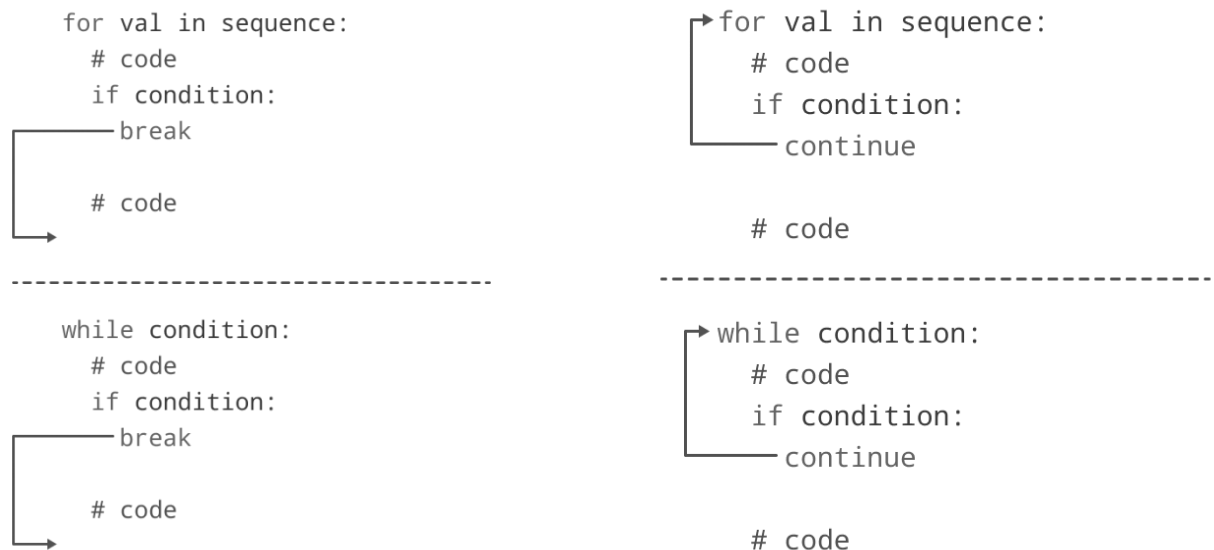
sum_of_digits.py > ...
1 num = int(input("Enter a number"))
2 sum_of_digits = 0
3 while num != 0:
4     rem = num % 10
5     sum_of_digits = sum_of_digits + rem
6     num = num // 10
7
8 print(f"Sum of digits = {sum_of_digits}")

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS E:\python_programming> python -u "e:\python_programming\sum_of_digits.py"
Enter a number512
Sum of digits = 8

```

7. Break and Continue



`break` and `continue` are control flow statements in Python used within loops to alter their behavior. The `break` statement is used to exit a loop prematurely. When encountered inside a loop, it immediately terminates the loop's execution and transfers control to the statement immediately following the loop. The `continue` statement is used to skip the rest of the code inside a loop for the current iteration and proceed to the next iteration of the loop.

```
break_contine.py > ...
1 # break and continue
2 for i in range(1, 11):
3     if i == 5:
4         break
5     if i % 2 == 0:
6         continue
7     print(i)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS E:\python_programming> python -u "e:\python_programming\break_contine.py"
1
3
```

Here, the code iterates through numbers from 1 to 10 using a for loop. Within the loop, the break statement is encountered when *i* equals 5, causing an immediate termination of the loop's execution. Therefore, only numbers from 1 to 4 are printed. Additionally, the continue statement is used to skip even numbers (if *i* % 2 == 0). When an even number is encountered, the loop skips the remaining code in the current iteration and proceeds to the next iteration. As a result, only odd numbers are printed, excluding 5 due to the preceding break statement. Thus, the output consists of odd numbers from 1 to 3.

8. Functions

A function is a block of code that performs a specific task. Dividing a complex problem into smaller chunks makes our program easy to understand and reuse.

Syntax:

```
def function_name(parameters):
    #statements
    return expression
```

def is a keyword, function_name must be an identifier.

Note: After creating a function in Python we can call it by using the name of the functions Python followed by parenthesis containing parameters of that particular function.

```
function_1.py > ...
1 # Defining a function to print hello world
2 def fun():
3     print("Hello world from function")
4
5 # Driver code to call a function
6 fun()

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS E:\python_programming> python -u "e:\python_programming\function_1.py"
Hello world from function
PS E:\python_programming> |
```

Here, the code defines a simple function named `fun` using the `def` keyword, which contains a single statement to print "Hello world from function". This function does not take any parameters and performs only this print operation. Following the function definition, the driver code calls the function `fun()` to execute its body. When `fun()` is called, the function prints the message "Hello world from function" to the console.

```
func_add.py > ...
1 # Defining a function with arguments
2 def sum_1(a, b):
3     print(f"The sum is {a+b}")
4
5 # Defining a function with arguments and return
6 def sum_2(a, b):
7     return a+b
8 # Driver code to call a function
9 sum_1(10,5)
10 print(f"The sum is {sum_2(20,40)}")

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS E:\python_programming> python -u "e:\python_programming\func_add.py"
The sum is 15
The sum is 60
```

Here, the code defines two functions to illustrate different ways of handling function arguments and return values. The first function, `sum_1(a, b)`, takes two arguments, `a` and `b`, and directly prints their sum using an f-string for formatted output. The second function, `sum_2(a, b)`, also takes two arguments, calculates their sum, but instead of printing it, it returns the result. In the driver code,

sum_1 is called with the arguments 10 and 5, resulting in the immediate print output "The sum is 15". The second function, sum_2, is called with the arguments 20 and 40, and its return value is printed using an f-string, resulting in the output "The sum is 60".

Q. WAP to check whether a number is prime or not

```
func_Prime_or_not.py > ...
1 # Defining a function to check prime or composite
2 def is_prime(num):
3     if num <= 1:
4         return "neither Prime nor Composite"
5     else:
6         for i in range(2, num//2 + 1):
7             if num % i == 0:
8                 return "Composite"
9         return "Prime"
10
11 # Driver code
12 num = int(input("Enter a number"))
13 print(f"{num} is {is_prime(num)}")

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS E:\python_programming> python -u "e:\python_programming\func_Prime_or_not.py"
Enter a number: 7
7 is Prime
PS E:\python_programming> |
```

Types of function arguments:

- Default argument: A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument.
- Keyword arguments (named arguments): The idea is to allow the caller to specify the argument name with values so that the caller does not need to remember the order of parameters.
- Positional arguments: A positional argument refers to an argument passed to a function or method based on its position or order in the function's parameter list. When calling a function, positional arguments are specified by their position in the argument list, and their values are matched with the parameters in the function definition according to their order.

```
keyword_args.py > ...
1 def division(a, b=10):
2     return a / b
3
4 print(division(10))
5 print(division(10, 20))
6 print(division(a=10, b=-30))
7 print(division(b=10, a=30))

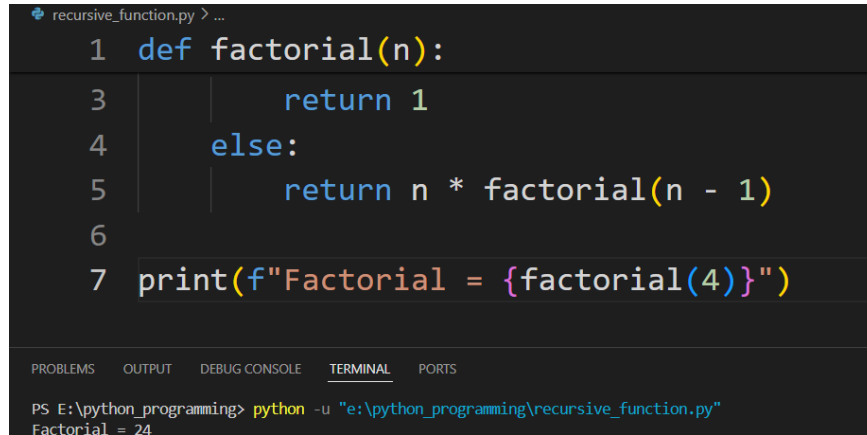
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS E:\python_programming> python -u "e:\python_programming\keyword_args.py"
1.0
0.5
-0.3333333333333333
3.0
```

Here, the function `division(a, b=10)` is defined with a default value of 10 for the parameter `b`. In the first print statement `print(division(10))`, the function is called with only one argument, resulting in 1.0 being printed, as 10 divided by the default value 10 equals 1.0. In the second print statement `print(division(10, 20))`, both arguments are provided explicitly, leading to 0.5 being printed, as 10 divided by 20 equals 0.5. In the third and fourth print statements `print(division(a=10, b=-30))` and `print(division(b=10, a=30))`, keyword arguments are used to specify values for `a` and `b`, overriding the default value for `b`. Therefore, the third statement prints -0.3333, while the fourth statement prints 3.0, reflecting the division of `a` by the explicitly provided `b` values.

9. Recursive Functions in Python

Recursion in Python refers to when a function calls itself. There are many instances when you have to build a recursive function to solve Mathematical and Recursive Problems.



```
recursive_function.py > ...
1 def factorial(n):
2
3     return 1
4 else:
5     return n * factorial(n - 1)
6
7 print(f"Factorial = {factorial(4)}")

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS E:\python_programming> python -u "e:\python_programming\recursive_function.py"
Factorial = 24
```

Here, the factorial function calculates the factorial of a given non-negative integer n . If n is 0, the function returns 1, as the factorial of 0 is defined as 1. Otherwise, the function recursively calls itself with the argument $n - 1$ and multiplies the result by n . This process continues until n becomes 0, at which point the recursion stops, and the accumulated product is returned. In the provided example, $\text{factorial}(4)$ is called, resulting in $4 * 3 * 2 * 1$, which evaluates to 24. In addition to calculating the factorial, this function illustrates the concept of a stack through recursion. When the factorial function is called with a non-zero value, it recursively calls itself with a decremented value of n . Each recursive call adds a new frame to the call stack, storing information about the function call, including the arguments and local variables. These frames are stacked on top of each other in memory, forming a stack structure. As each recursive call reaches the base case ($n == 0$), it begins to return values back up the call stack. The returned values are multiplied together, ultimately yielding the factorial of the original input. Once the base case is reached, the stack begins to unwind, with each function call returning its computed result until the original call to $\text{factorial}(4)$ receives the final result.

Q. WAP to calculate Fibonacci series using Recursion

Q. WAP to calculate power of a number using Recursion.

Lab Questions

1. Write a program in Python to calculate the area of a circle given its radius.
2. Write a program in Python to convert Celsius to Fahrenheit.
3. Write a program in Python to check whether a given year is a leap year or not.
4. Write a program in Python to reverse a given string.
5. Write a program in Python to reverse a number.
6. Write a program in Python to find the factorial of a given number using recursion.
7. Write a program in Python to generate Fibonacci series up to n terms.
8. Write a program in Python to count the number of vowels in a given string.
9. Write a program in Python to find the sum of digits of a number.
10. Write a program in Python to check whether a given number is prime or not.
11. Write a program in Python to find the length of a string without using the built-in function.
12. Write a program in Python to print the pattern:
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
13. Write a program in Python to check whether a given string is a palindrome or not.
14. Write a program in Python to find the GCD (Greatest Common Divisor) of two numbers.
15. Write a program in Python named `find_max` that takes two numbers as input and returns the maximum of the two.
16. Write a program in Python named `is_prime` that takes a number as input and returns `True` if it is a prime number, otherwise returns `False`.
17. Write a program in Python named `check_palindrome` that takes a string as input and returns `True` if it is a palindrome, otherwise returns `False`.
18. Write a program in Python named `calculate_factorial` that takes a positive integer as input and returns its factorial without using recursion.