While loop, Nested Loops and Functions

Overview

In this session, participants will learn the concepts of while loop, understanding its syntax and practical examples. We'll merge loops with conditionals, enabling richer program flows, and discover the break and continue statements, mastering control within loops. Also, the python functions and lambda functions with examples.

Objectives:

1. Working with while loop: syntax, conditions, and examples
2. Combining loops and conditionals
3. Using the break statement to exit loops prematurely.
4. Utilizing the continue statement to skip iterations.
5. Implementing nested loops for complex iterations
6. Introduction to functions: purpose, advantages, and best practices
7. Defining and calling user-defined functions
8. Parameters and arguments: positional, keyword, and default values
9. Lambda function
10. Type Casting Python
11. Variable scope and lifetime

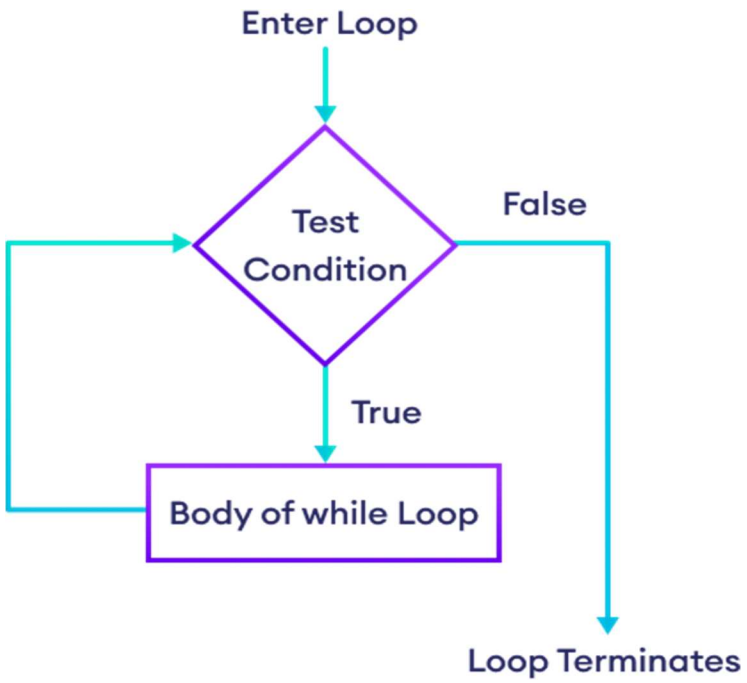
1. While loop: syntax, conditions, and examples

Python while loop is used to run a block of code until a certain condition is met.

Syntax:

```
while [enter the condition that the loop proceeds]:
    [what is done in the loop]
```

1. A while loop evaluates the condition.
2. If the condition evaluates to True, the code inside the while loop is executed.
3. Condition is evaluated again.
4. The process continues until the condition is False.
5. When the condition evaluates to False, the loop stops.



Example:

```
x = 0
while (x < 10):
    x = x + 1
    print (x)
```

Output:

```
1 2 3 4 5 6 7 8 9
```

Example: Adding elements to a list using while loop

```
myList = []
i = 0
while len(myList) < 4:
    myList.append(i)
    i += 1
print(myList)
```

Output:

```
[0, 1, 2, 3]
```

1.1 Combining Loops and Conditionals

It's like a daily routine: Every day (loop) if it's sunny (conditional), you go for a walk. Otherwise, you stay in.

While loop with else:

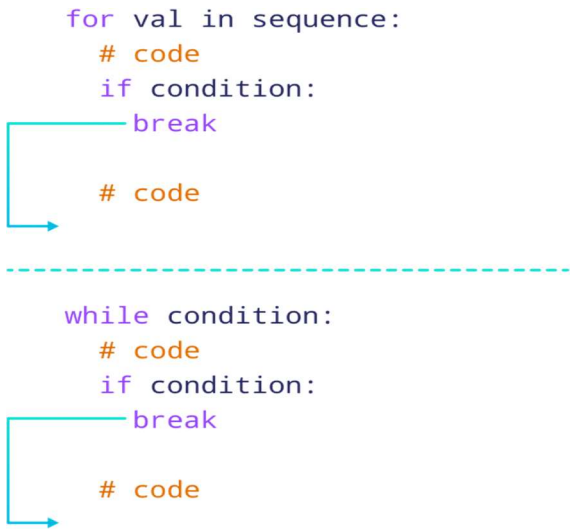
```
counter = 0
while counter < 3:
  print('Inside loop')
  counter = counter + 1
else:
  print('Inside else')
```

Output:

```
Inside loop
Inside loop
Inside loop
Inside else
```

1.2 Using the break Statement to Exit Loops Prematurely

With the break statement we can stop the loop even if the loop condition is true:



Example:

```
for i in range(5):
  if i == 3:
     break
  print(i)
```

Output:

```
0
1
2
```

Example:

```
i = 1
while i <= 10:
   print('6 *',i, '=',6 * i)
      if i == 5:
          break
   i = i + 1
```

Output:

```
6 * 1 = 6
6 * 2 = 12
6 * 3 = 18
6 * 4 = 24
6 * 5 = 30
```

1.3 Utilizing the continue Statement to Skip Iterations

With the continue statement we can stop the current iteration, and continue with the next:

```
for val in sequence:
    # code
    if condition:
        continue

    # code
```

```
while condition:
    # code
    if condition:
        continue

    # code
```

Example:

```
for i in range(5):
      if i == 3:
          continue
   print(i)
```

Output:

```
0
1
2
4
```

Example:

```
for letter in 'Python':
    if letter == 'h':
        continue
    print('Current Letter :', letter)
```

Output:

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
```

1.4 Implementing Nested Loops for Complex Iteration

A nested loop is a loop inside the body of the outer loop. The inner or outer loop can be any type, such as a while loop or for loop. For example, the outer for loop can contain a while loop and vice versa. For each iteration of an outer loop the inner loop re-starts and completes its execution before the outer loop can continue to its next iteration.

Example:

```
x = [1, 2]
y = [4, 5]
for i in x:
    for j in y:
      print(i, j)
```

Output:

```
1 4
1 5
2 4
2 5
```

Example:

```
lists = [["apple","mango","banana"], [1, 2, 3, 4 ,5], [True, False,
False]]

            for item in lists:
                  for i in range(len(item)):
                     print(item[i], end =' ')
                  print('')
```

Output:

```
apple mango banana
1 2 3 4 5
True False False
```

Example:

```
                for i in range(1,5):
                    for j in range(i):
                       print(i, end='')
                    print()
```

Output:

1

2 2

3 3 3

4 4 4 4

Explanation:

Iteration i	Iteration j	Output
1	1	1 (newline added after loop)
2	1	2
	2	2 (newline added after loop)
3	1	3
	2	3
	3	3 (newline added after loop)
4	1	4
	2	4
	3	4
	4	4 (newline added after loop)

2. Introduction to Functions: Purpose, Advantages, and Best Practices

In Python, functions are defined blocks of reusable code. Their primary advantages include modularizing code, promoting reusability, and enhancing clarity and maintainability. Employing functions effectively can lead to more efficient debugging and improved code organization.

2.1 Defining and Calling User-Defined Functions

Function is a block of code which only runs when it is called. You can pass data, known as arguments, into a function. A function can return data as a result.

There are two types of function in Python programming:

1. Standard library functions - These are built-in functions in Python that are available to use.

2. User-defined functions - We can create our own functions based on our requirements.

User defined functions are defined using the **def** keyword followed by the function name. The code within the function is executed when the function is called. To call a function, you use its name followed by parentheses: **function_name().**

Syntax:

```
def function_name (arguments):
#body of function
return
```

Example: A function without arguments and return parameter.

```
def greet ():
   print("Hello World")
greet()#call function
```

Output:

```
Hello World
```

2.2 Parameters and Arguments: Positional, Keyword, and Default Values

Parameters are named entities in a function definition that specify what input the function can accept. When calling a function, you pass values (arguments) to these parameters. Positional arguments are read in order, keyword arguments are explicitly tied to parameter names using an equals (=) sign, and default parameters provide fallback values within the function definition.

- def – keyword used to declare a function
- function_name – any name given to a function
- arguments – any value passed to function
- return – keyword (optional) returns value from a function

Example: A function with one argument.

```
def print_name(first_name):
      print(first_name)
#Call function
print_name("Sara")
```

Output:

```
Sara
```

Example: A function with two arguments.

```
def print_name(first_name, last_name):
   print(first_name, last_name)
#Call function
print_name("Sara", "Ali")
```

Output:

```
Sara Ali
```

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you must call the function with 2 arguments, not more, and not less.

Example: Sending arguments with *key = value* syntax.

```
def print_name(first_name, last_name):
   print(first_name, last_name)
#call function
print_name(first_name = "Sara", last_name = "Ali")
```

Example: A function with a return statement.

```
def add_numbers(num1,num2):
      sum = num1 + num2
     return sum   #statement inside func hence indentation.
#call function
result = add_numbers(5, 10) print(result)
```

Output:

15

2.3 Default Parameter Value

If a function is called without the arguments, then default value is used. The following example illustrates how to use default values.

Example: A function with default parameters.

```
def add_numbers(num1 = 0,num2 = 0):
      sum = num1 + num2
      return sum
# Call function
print(add_numbers(5, 10))
print(add_numbers(5))
print(add_numbers())
```

Output:

```
15
5
0
```

2.4 Python Function with Arbitrary Arguments

Sometimes, we do not know in advance the number of arguments that will be passed into a function. To handle this kind of situation, we can use arbitrary arguments in Python. Arbitrary arguments allow us to pass a varying number of values during a function call. We use an asterisk (*) before the parameter name to denote this kind of argument.

Example: A function with default parameters.

```
def find_sum(*numbers):
    result = 0
    for num in numbers:
        result = result + num
    print("Sum = ", result)


# function call with 3 arguments find_sum(1, 2, 3)

# function call with 2 arguments find_sum(4, 9)
```

Output:

```
Sum =  6
Sum =  13
```

2.5 Python Lambda

A lambda is a small anonymous function. It can take any number of arguments but has only one expression. The expression is executed, and the result is returned.

Syntax: **lambda *arguments: expression***

Example:

```
x = lambda a: a + 10

print(x(5))
```

Output: 15

2.6 Difference Between Lambda functions and def defined function

With lambda function	Without lambda function
Supports single-line sometimes statements that return some value.	Supports any number of lines inside a function block
Good for performing short operations/data manipulations.	Good for any cases that require multiple lines of code.
Using the lambda function can sometime reduce the readability of code.	We can use comments and function descriptions for easy readability.

Example:

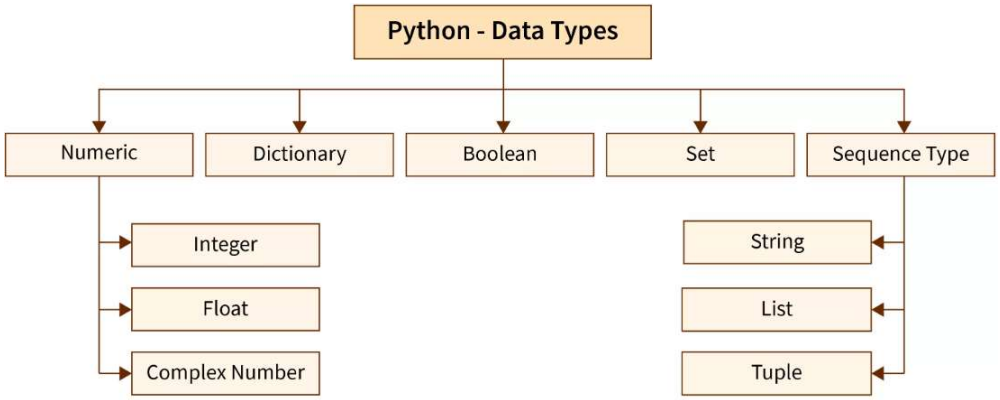
```
def cube(y):
    return y*y*y

lambda_cube = lambda y: y*y*y

# using function defined
# using def keyword
print("Using function defined with `def` keyword, cube:", cube(5))

# using the lambda function
print("Using lambda function, cube:", lambda_cube(5))
```

Output:

```
Using function defined with `def` keyword, cube: 125
Using lambda function, cube: 125
```

2.7 Type Casting Python

Type casting is the process of converting one data type into another. For example, converting int data type to string datatype.



2.7.1 Implicit Type Conversion:

In certain situations, Python automatically converts one data type into another. This is known as implicit type conversion.

Example:

```
integer_number = 123
float_number = 1.23

new_number = integer_number + float_number

# display new value and resulting data type
print("Value:",new_number)
print("Data Type:",type(new_number))
```

Output:

```
Value: 124.23
Data Type: <class 'float'>
```

Python always converts smaller data types to larger data types to avoid the loss of data.

2.7.2 Explicit Type Conversion:

In Explicit Type Conversion, users convert the data type of an object to the required data type.

- int(): constructs an integer number from an integer literal, a float literal (by removing all decimals), or a string literal (providing the string represents a whole number)
- float(): constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
- str(): constructs a string from a wide variety of data types, including strings, integer literals and float literals
- bool(): constructs a Boolean from a variety of data types including integers,strings, and floats

```
a = int(1) # a will be 1
b = int(2.8) # b will be 2
c = int("3") # c will be 3
d = float(1)     # d will be 1.0
e = float(2.8)   # e will be 2.8
f = float("3")   # f will be 3.0
g = float("4.2") # g will be 4.2
h = str("s1") # h will be 's1'
i = str(2) # i will be '2'
j = str(3.0) # j will be '3.0'
k = bool(-1) # k will be True
l = bool(0) # l will be False
m = bool(-6.0) # m will be True
```

2.8 Variable Scope and Lifetime

Using a function defines the concept of scope in Python.

If we define a variable inside a function or pass a value to a function's parameter, the value of that variable is only accessible within that function. This is called Scope.

In other words, the scope is the region of a program where we can access a particular identifier or variable. This is called scope of variable or simply variable scope.

Variables declared inside a program may not be accessible at all locations of that program. It depends on the where we have declared a variable in a program.

Variables in Python have two main scopes:

Local Scope: Variables defined inside a function and can't be accessed outside of it.

Global Scope: Variables defined outside all functions and can be accessed and modified throughout the module. To modify a global variable inside a function, you must use the global keyword.

Example:

```
# Creating a variable city and set to New York.
city = 'New York' # global scope.
def showMe():
    city = 'Dhanbad' # local scope within a function.
    print('City:',city)
# Function call
showMe()
print('City:',city) # accessing global variable from the global scope.
```

Output:

```
City: Dhanbad
City: New York
```

Practice Questions:

1. Use a while loop to create a program that keeps asking the user for their name until they type "stop".
2. Write a script that combines while loops and conditionals to print numbers from 1 to 20, but skips numbers that are multiples of 3.
3. Create a for loop that goes through numbers 1 to 50. If a number is a multiple of 10, use the break statement to exit the loop.
4. Consider the list:
   fruit = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
   Use a loop to search the fruit “kiwi” from the list fruit. Break the loop once the fruit is found.
5. Use nested for loop to display the following pattern.
6. Use a while loop to keep asking for a number until a negative number is entered. At the end, print the sum of all entered numbers.
7. Define a function called greet that takes a name as a parameter and prints a greeting using that name.
8. Write a function called calculate_area that takes the length and width of a rectangle and returns its area.
9. Write a Python function to find the max of three numbers.
10. Define a lambda function that returns a square of a number. Now, call the lambda function in a loop to square values in the following list:
    list_numbers = [2, 4, 5, 8, 9, 12] Display the list with squared items.