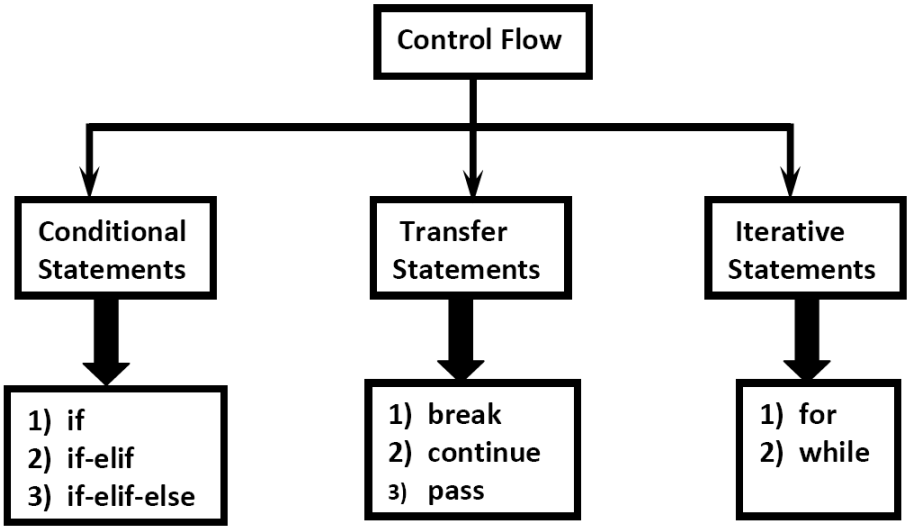
## UNIT-2: Decision-Making Statements, Loops and User-Defined Functions

Conditional Statements; While loop, for loop; range () function, nested loops; While-else, For-else, break, continue, pass;
Functions: Syntax and basics of function and usage; Passing Parameters, arguments in a function – Default, keyword, positional and Variable - length arguments; local and global scope of variable; return statement, recursive function, recursion vs iteration;

## 2.0 Control Flow Statements:

- Control Flow describes the order in which statements will be executed at runtime.
- Statements that allow the computer to select or repeat an action.

Control Flow statements are divided into three categories in Python.




### 2.0.1. Conditional/Control Statements (or) Selection Statements -

When there is no condition placed before any set of statements, the program will be executed in sequential manure. But when some condition is placed before a block of statements the flow of execution might change depends on the result evaluated by the condition. This type of statement is also called decision making statements or control statements. This type of statement may skip some set of statements based on the condition.

Based on some condition result, some group of statements will be executed and some group of statements will not be executed.

**Note:**

1. There is no **switch** statement in Python. (Which is available in C and Java)
2. There is no **do-while** loop in Python. (Which is available in C and Java)
3. **goto** statement is also not available in Python. (Which is available in C)

**Logical Conditions Supported by Python:**

a) Equal to (==) **Example:** a == b
b) Not Equal (! =) **Example:** a! = b
c) Greater than (>) **Example:** a > b
d) Greater than or equal to (>=) **Example:** a >= b
e) Less than (<) **Example:** a < b
f) Less than or equal to (<=) **Example:** a <= b

**Indentation:**

To represent a block of statements other programming languages like C, C++ uses "{…}" curly – brackets, instead of this curly braces python uses indentation using white space which defines scope in the code. The example given below shows the difference between usage of Curly bracket and white space to represent a block of statement.

| C Program | Python |
|---|---|
| x = 500<br>y = 200<br>if (x > y)<br>{<br>printf("x is greater than y")<br>}<br>else if(x == y)<br>{<br>printf("x and y are equal")<br>}<br>else<br>{<br>printf("x is less than y") | x = 500<br>y = 200<br>if x > y:<br>  print("x is greater than y")<br>elif x == y:<br>  print("x and y are equal")<br>else:<br>  print("x is less than y") |

**Without proper Indentation:**

```
In [14]: if 10<20:
         print('10 is less than 20')
         print('End of Program')
```

```
  Input In [14]
    print('10 is less than 20')
    ^
IndentationError: expected an indented block
```

**2.0.1 if Statement:** The "if" statement is written using "if" keyword, followed by a condition. If the condition is true the block will be executed. Otherwise, the control will be transferred to the first statement after the block.

**Syntax:**

if condition: (**Note:** In Python anywhere we are using colon (:) means we are defining block)

statement 1

statement 2

statement 3 (This statement is not under if statement)

**Example1:**

```
In [7]: x = 200
        y = 100
        if x > y:
           print("x is greater than y")
        print("End")
```

```
x is greater than y
End
```

**Example2:**

```
In [*]: name=input("Enter Name:")
        if name=="Leela Prasad":
           print("Hello Leela Prasad Good Morning")
        print("How are you!!!")
```

```
Enter Name:
```

```
In [*]: name=input("Enter Name:")
        if name=="Leela Prasad":
           print("Hello Leela Prasad Good Morning")
        print("How are you!!!")
```

```
Enter Name: Leela Prasad
```

```
In [13]: name=input("Enter Name:")
         if name=="Leela Prasad":
            print("Hello Leela Prasad Good Morning")
         print("How are you!!!")
```

```
Enter Name:Leela Prasad
Hello Leela Prasad Good Morning
How are you!!!
```

### 2.0.2 if - else Statement:

**Syntax:**

if condition:

Action 1

else:

Action 2

if condition is true then Action-1 will be executed otherwise Action-2 will be executed.

**Example1:**

```
In [3]: name = input('Enter Name : ')
        if name == 'Karthi':
           print('Hello Karthi! Good Morning')
        else:
           print('Hello Guest! Good MOrning')
        print('How are you?')
```

```
Enter Name : Karthi
Hello Karthi! Good Morning
How are you?
```

**Example2:**

```
In [1]: name = input('Enter Name : ')
        if name == 'Karthi':
           print('Hello Karthi! Good Morning')
        else:
           print('Hello Guest! Good MOrning')
        print('How are you?')
```

```
Enter Name : Prasad
Hello Guest! Good MOrning
How are you?
```

### 2.0.3 if-elif-else Statement:

**Syntax:**

if condition1:

Action-1

elif condition2:

Action-2

elif condition3:

Action-3

elif condition4:

Action-4

...

else:

Default Action

Based condition the corresponding action will be executed.

**Example:**

```
In [1]: brand=input("Enter Your Favourite Brand:")
        if brand=="RC":
            print("It is childrens brand")
        elif brand=="KF":
            print("It is not that much kick")
        elif brand=="FO":
            print("Buy one get Free One")
        else :
            print("Other Brands are not recommended")
```

```
Enter Your Favourite Brand:RC
It is childrens brand
```

## 2.0.4 Example Programs on Conditional Statements:

**1. Write a program to find biggest of given 2 numbers.**

```
n1=int(input("Enter First Number:"))
n2=int(input("Enter Second Number:"))
if n1>n2:
    print("Biggest Number is:",n1)
else :
    print("Biggest Number is:",n2)
```

```
Enter First Number: 20
Enter Second Number: 30
Biggest Number is: 30
```

**2. Write a program to find biggest of given 3 numbers.**

```
n1=int(input("Enter First Number:"))
n2=int(input("Enter Second Number:"))
n3=int(input("Enter Third Number:"))
if n1>n2 and n1>n3:
    print("Biggest Number is:",n1)
elif n2>n3:
    print("Biggest Number is:",n2)
else:
    print("Biggest Number is:",n3)
```

```
Enter First Number: 80
Enter Second Number: 45
Enter Third Number: 91
Biggest Number is: 91
```

**3. Write a program to find smallest of given 2 numbers?**

```
n1=int(input("Enter First Number:"))
n2=int(input("Enter Second Number:"))
if n1>n2:
    print("Smallest Number is:",n2)
else:
    print("Smallest Number is:",n1)
```

```
Enter First Number: 50
Enter Second Number: 25
Smallest Number is: 25
```

**4. Write a program to find smallest of given 3 numbers?**

```
n1=int(input("Enter First Number:"))
n2=int(input("Enter Second Number:"))
n3=int(input("Enter Third Number:"))
if n1<n2 and n1<n3:
    print("Smallest Number is:",n1)
elif n2<n3:
    print("Smallest Number is:",n2)
else:
    print("Smallest Number is:",n3)
```

```
Enter First Number: 10
Enter Second Number: 15
Enter Third Number: 18
Smallest Number is: 10
```

**5. Write a program to check whether the given number is even or odd?**

```
n1=int(input("Enter First Number:"))
rem = n1 % 2
if rem == 0:
    print('Entered Number is an Even Number')
else:
    print('Entered Number is an Odd Number')
```

```
Enter First Number: 17
Entered Number is an Odd Number
```

**6. Write a program to check whether the given number is in between 1 and 100?**

```
n=int(input("Enter Number:"))
if n>=1 and n<=100 :
    print("The number",n,"is in between 1 to 100")
else:
    print("The number",n,"is not in between 1 to 100")
```

```
Enter Number: 18
The number 18 is in between 1 to 100
```

**7. Write a program to take a single digit number from the key board and print it's value in English word?**

```
n=int(input("Enter a digit from o to 9:"))
if n==0 :
    print("ZERO")
elif n==1:
    print("ONE")
elif n==2:
    print("TWO")
elif n==3:
    print("THREE")
elif n==4:
    print("FOUR")
elif n==5:
    print("FIVE")
elif n==6:
    print("SIX")
elif n==7:
    print("SEVEN")
elif n==8:
    print("EIGHT")
elif n==9:
    print("NINE")
else:
    print("PLEASE ENTER A DIGIT FROM 0 TO 9")
```

```
Enter a digit from o to 9: 9
NINE
```

## 2.1 Iterative Statements:

Sometimes certain section of the code (block) may need to be repeated again and again as long as certain condition remains true. In order to achieve this, the iterative statements are used. The number of times the block needs to be repeated is controlled by the test condition used in that statement. This type of statement is also called as the "Looping Statement". Looping statements add a surprising amount of new power to the program.

### 2.1.1 Need for Looping / Iterative Statement –

Suppose the programmer wishes to display the string "Prasad!..." 150 times. For this, one can use the print command 150 times.

print("Prasad!...")

print("Prasad!...")

…..

The above method is somewhat difficult and laborious. The same result can be achieved by a loop using just two lines of code.

**Example:**

```
for count in range(1,150):
        print ("Prasad")
```

Python supports 2 types of iterative statements.

1. for loop
2. while loop

## 2.1.11. The 'for' Loop:

The **for loop** is one of the powerful and efficient statements in python which is used very often. It specifies how many times the body of the loops needs to be executed. For this reason, it uses control variables which keep tracks, the count of execution.

**The general syntax of a "for" loop looks as below:**

**for** value in sequence:

  body of the loop

where 'sequence' can be string or any collection. Body will be executed for every element present in the sequence.

**Example 1:** To compute the sum of first n numbers (i.e. 1 + 2 + 3 + ……. + n)

```
In [3]: total = 0
        n = int (input ("Enter a Positive Number"))
        for i in range(1,n+1):
            total = total + i
        print ("The Sum is ", total)
```

```
Enter a Positive Number4
The Sum is  10
```

In the above program, the statement total = total + i is repeated again and again „n" times. The number of execution count is controlled by the variable „i". The range value is specified earlier before it starts executing the body of loop. The initial value for the variable i is 1 and final value depends on „n". You may also specify any constant value.

**Example 2:** to find the sum of squares of each element of the list using for loop

```
In [7]: # creating the list of numbers
        numbers = [3, 5, 23, 6, 5, 1, 2, 9, 8]

        # initializing a variable that will store the sum
        sum = 0

        # using for loop to iterate over the list
        for num in numbers:

            sum = sum + num ** 2

        print("The sum of squares is: ", sum)
```

```
The sum of squares is:  774
```

**Example 3:** Write a Program to print characters present in the given string.

```
In [11]: s="Prasad"
         for x in s :
             print(x)
```

```
P
r
a
s
a
d
```

## 2.1.12 The for ….. else:

The for-else statement is used to execute a block of code once the for loop completes its iteration over a sequence, provided that the loop was not terminated by a break statement. The else block will not be executed if the for loop is exited via break.

**Syntax:**

for item in sequence:

    # Loop code block

    if condition:

        break

else:

    # Else code block

    # This block executes only if the loop completes without encountering a break statement

**Example:**

```
for i in range(5):
    print(i)
else:
    print("Loop completed without a break.")
```

```
0
1
2
3
4
Loop completed without a break.
```

## 2.1.13 The range() Function:

The range() function can be called in three different ways based on the number of parameters. All parameter values must be integers.

| Type | Example | Explanation |
|---|---|---|
| range(end) | for i in range(5):<br>print(i)<br>**Output:**<br>0,1,2,3,4 | This begins at 0. Increments by 1. End just before the value of end parameter. |
| range(begin, end) | for i in range(2,5):<br>print(i)<br>**Output:**<br>2,3,4 | Starts at begin, End before end<br>value, Increment by 1 |
| range(begin, end, step) | for i in range(2,7,2)<br>print(i)<br>**Output:**<br>2,4,6 | Starts at begin, End before end<br>value, increment by step value |

**Table: Categories of range function**

**Example:** To find the Factorial of a number “n"

```
In [20]: n= int(input("Enter a Number :"))
factorial = 1
if n < 0:
    print("Negative Number , Enter valid Number !...")
elif n == 0:
    print("The factorial of 0 is 1")
else:
    for i in range(1, n + 1):
        factorial = factorial*i
print("The factorial of" ,n, "is", factorial)
```

```
Enter a Number :3
The factorial of 3 is 6
```

## 2.1.14 while loop:

If we want to execute a group of statements iteratively until some condition false, then we should go for while loop.

**Syntax:**

while condition:

body

**Example 1:** To print numbers from 1 to 10 by using while loop

```
In [1]: x=1
        while x <=10:
            print(x)
            x=x+1
```

```
1
2
3
4
5
6
7
8
9
10
```

**Example 2:** To display the sum of first n numbers.

```
In [2]: n=int(input("Enter number:"))
        sum=0
        i=1
        while i<=n:
            sum=sum+i
            i=i+1
        print("The sum of first",n,"numbers is :",sum)
```

```
Enter number:5
The sum of first 5 numbers is : 15
```

**Example 3:** write a program to prompt user to enter some name until entering Prasad.

```
In [3]: name=""
        while name!="Prasad":
            name=input("Enter Name:")
        print("Thanks for confirmation")
```

```
Enter Name:Leela
Enter Name:Prasad
Thanks for confirmation
```

## 2.1.15 The while ... else Loop:

the while-else statement is a control flow construct that combines a while loop with an else clause. The else block is executed if the while loop terminates normally, that is, without encountering a break statement.

**Syntax:**

while condition:

    # code to be executed while the condition is True

else:

    # code to be executed if the loop terminates normally

**Example:**

```
x = 0
while x < 5:
    print(x)
    x += 1
else:
    print("Loop completed without a break.")
```

```
0
1
2
3
4
Loop completed without a break.
```

In this example, the while loop iterates as long as i is less than 5. After the loop completes, the else block is executed, printing the message "The loop has finished normally."

## 2.2 Nested Loops:

In a nested loop, the inner loop completes all its iterations for every single iteration of the outer loop. This structure is useful for performing operations on each element of a multi-dimensional array or list.

**Syntax:**

The basic syntax of nested loops in Python is:

for outer_variable in outer_sequence:

    for inner_variable in inner_sequence:

        # Code block to execute for each iteration of inner_variable

    # Code block to execute for each iteration of outer_variable

**Example 1:**

```
In [6]: for i in range(3):
            for j in range(2):
                print('Hello')
```

```
Hello
Hello
Hello
Hello
Hello
Hello
```

**Example 2:**

```
In [10]: for i in range(2):
             for j in range(4):
         #print("i=",i," j=",j)
                 print('i = {} j = {}'.format(i,j))
```

```
i = 0 j = 0
i = 0 j = 1
i = 0 j = 2
i = 0 j = 3
i = 1 j = 0
i = 1 j = 1
i = 1 j = 2
i = 1 j = 3
```

**Example 3:** Write a program to dispaly *'s in Right angled triangled form.

```
In [21]: n = int(input("Enter number of rows:"))
         for i in range(1,n+1):
             for j in range(1,i+1):
                 print("*",end=" ")
             print()
```

```
Enter number of rows:8
*
* *
* * *
* * * *
* * * * *
* * * * * *
* * * * * * *
* * * * * * * *
```

## 2.3 The Break Statement:

The **break** statement in Python terminates the current loop and resumes execution at the next statement, just like the traditional break found in C. The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The break statement can be used in both while and for loops.

**Example 1:**

```
In [2]: for i in range(10):
            if i==7:
                print("processing is enough..plz break")
                break
        print(i)
```

```
processing is enough..plz break
7
```

**Example 2:**

```
In [4]: for letter in 'Python':
            if letter == 'h':
                break
        print('Current Letter :', letter)
```

```
Current Letter : h
```

**Example 3:**

```
In [7]: var = 10
        while var > 0:
            print('Current variable value :', var)
            var = var -1
            if var == 5:
                break
        print("Good bye!")
```

```
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Good bye!
```

## 2.4 The Continue Statement:

The continue statement is used to skip the current iteration of the loop and proceed with the next iteration. It does not terminate the loop; instead, it skips the remaining code in the current iteration and jumps to the beginning of the next iteration.

**Example 1:** To print odd numbers in the range 0 to 9.

```
In [11]: for i in range(10):
             if i%2==0:
                 continue
             print(i)
```

```
1
3
5
7
9
```

**Example 2:**

```
In [10]: for letter in 'Python':
             if letter == 'h':
                 continue
         print('Current Letter :', letter)
```

```
Current Letter : n
```

**Example 3:**

```
In [13]: var = 10
         while var > 0:
             var = var -1
             if var == 5:
                 continue
             print('Current variable value :', var)
         print("Good bye!")
```

```
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Current variable value : 0
Good bye!
```

## 2.5 The pass Statement:

The pass statement is a placeholder that does nothing. It is used in places where a statement is syntactically required, but no action is needed. It is often used as a placeholder for future code.

**Syntax:** pass

**Example:**

```
for i in range(100):
    if i%9==0:
        print(i)
    else:
        pass
```

```
0
9
18
27
36
45
54
63
72
81
90
99
```

## Summary:

- **break:** Terminates the loop and exits immediately.
- **continue:** Skips the rest of the current iteration and moves to the next iteration.
- **pass:** Does nothing and serves as a placeholder.

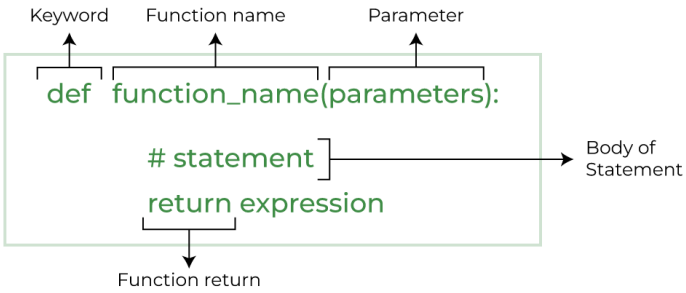
These control flow statements provide flexibility in managing loops and conditional statements, allowing you to control the flow of your program effectively.

## 2.5 Basics of Function and Usage:

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

**Usage:** To put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code again and again for different inputs, we can do the function calls to reuse code contained in it over and over again.

**Syntax:**




### 2.5.1 Creating a Function:

In Python a function is defined using the def keyword:

**Example:**

```
def add_numbers(a, b):
    """This function adds two numbers."""
    result = a + b
    return result

# Calling the function
sum = add_numbers(3, 5)
print(sum)
```

```
8
```

### 2.5.2 Calling a Function:

After creating a function, we can call it by using the name of the function followed by parenthesis containing parameters of that particular function.

**Example:**

```
In [4]: def calculate(a,b):
            print('The Sum : ', a + b)
            print('The Difference : ', a - b)
            print('The Product : ', a * b)
        a = 20
        b = 10
        calculate(a,b)
        a = 200
        b = 100
        calculate(a,b)
```

```
The Sum :  30
The Difference :  10
The Product :  200
The Sum :  300
The Difference :  100
The Product :  20000
```

## 2.5.3 return values of function:

The return statement in Python is used to exit a function and send a value back to the caller. It is a fundamental aspect of functions that enables you to get results from the computations performed inside the function.

**a) Basic Usage**

The return statement is typically used at the end of a function. When the return statement is executed, the function terminates and the value specified by return is sent back to the caller.

**Example:**

```
def sum( arg1, arg2 ):
    total = arg1 + arg2
    print("Inside the function : ", total)
    return total;
# Now you can call sum function
total = sum( 10, 20 );
print("Outside the function : ", total)
```

```
Inside the function :  30
Outside the function :  30
```

**b) Multiple Return Statements**

A function can have multiple return statements. The function will exit as soon as a return statement is executed.

**Example:**

```
def check_number(num):
    if num > 0:
        return "Positive"
    elif num < 0:
        return "Negative"
    else:
        return "Zero"

print(check_number(10))
print(check_number(-5))
print(check_number(0))
```

```
Positive
Negative
Zero
```

## 2.5.3 Parameter Passing:

A parameter is a variable in the declaration of a function. Parameters are used to define the inputs that a function can accept. They act as placeholders for the values that will be passed to the function when it is called.

**Example:**

```
def greet(name, age):  # name and age are parameters
    print(f"Hello, {name}! You are {age} years old.")
```

## 2.5.4 Arguments in function:

In Python, arguments are values that you pass into a function when you call it. There are different types of arguments in python functions

### 1. Positional Arguments:

Positional arguments are the most straightforward way to pass values to a function. The arguments are matched to the parameters based on their position.

- **Example:**

```
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")

greet("Alice", 30)  # Output: Hello, Alice! You are 30 years old.
```

```
Hello, Alice! You are 30 years old.
```

### 2. Keyword (i.e., Parameter name) Arguments:

Keyword arguments allow you to pass arguments using the parameter names. This makes the function call more readable and allows you to pass arguments in any order.

- **Example:**

```
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")

greet(age=40, name="Charlie")  # Output: Hello, Charlie! You are 40 years old.
```

```
Hello, Charlie! You are 40 years old.
```

### 3. Default Arguments:

Default arguments are used to specify default values for parameters. If no value is provided for a default parameter, the default value is used.

- **Example:**

```
def greet(name, age=25):
    print(f"Hello, {name}! You are {age} years old.")

greet("Bob")  # Output: Hello, Bob! You are 25 years old.
```

```
Hello, Bob! You are 25 years old.
```

### 4. Variable length Arguments:

- Variable-length arguments allow you to pass an arbitrary number of positional or keyword arguments using *args and **kwargs.
- *args is used to pass a variable number of non-keyword arguments to a function. It collects all the positional arguments into a tuple. This is useful when you don't know in advance how many arguments will be passed to the function.
- **Example:**

```
def add_numbers(*args):
    total = 0
    for num in args:
        total += num
    return total

# Calling the function with different numbers of arguments
print(add_numbers(1, 2, 3))      # Output: 6
print(add_numbers(5, 10, 15, 20))  # Output: 50
```

```
6
50
```

**In this example**, the **function add_numbers** can accept any number of positional arguments, and they are all collected into the **args** tuple.

- **kwargs is used to pass a variable number of keyword arguments to a function. It collects all the keyword arguments into a dictionary.
- **Example:**

```
def display_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

# Calling the function with different keyword arguments
display_info(name="Alice", age=30, city="New York")
```

```
name: Alice
age: 30
city: New York
```

**In this example,** the **function display_info** can accept any number of keyword arguments, and they are all collected into the **kwargs** dictionary.

## 2.6 Local and Global Scope of a Variable:

A variable is only available from inside the region it is created. This is called scope.

### 2.6.1 Local Scope:

A variable created inside a function belongs to the local scope of that function, and can only be used inside that function.

**Example:**

A variable created inside a function is available inside that function:

```
In [16]: def myfunc():
           x = 300
           print(x)

         myfunc()
```

```
300
```

### a) Function inside Function:

A variable created inside a function belongs to the local scope of that function, and can only be used inside that function, the variable x is not available outside the function, but it is available for any function inside the function:

**Example:**

The local variable can be accessed from a function within the function:

```
In [17]: def myfunc():
           x = 300
           def myinnerfunc():
             print(x)
           myinnerfunc()

         myfunc()
```

```
300
```

### 2.6.2 Global Scope:

A variable created in the main body of the Python code is a global variable and belongs to the global scope.

Global variables are available from within any scope, global and local.

**Example:**

A variable created outside of a function is global and can be used by anyone:

```
In [18]: x = 300

         def myfunc():
           print(x)
         myfunc()
         print(x)

         300
         300
```

**a) Naming Variables:**

If you operate with the same variable name inside and outside of a function, Python will treat them as two separate variables, one available in the global scope (outside the function) and one available in the local scope (inside the function):

**Example:**

The function will print the local x, and then the code will print the global x:

```
In [26]: def myfunc():
           x = 200
           print(x)

         myfunc()
         print(x)

         200
         300
```

**b) Global Keyword:**

The global keyword in Python is used to declare that a variable inside a function is global. This means that any changes made to the variable inside the function are reflected outside the function. Without the global keyword, the variable is treated as local within the function, and changes to it do not affect the global variable.

The global keyword makes the variable global.

**Example:**

If you use the global keyword, the variable belongs to the global scope:

```
In [29]: def myfunc():
           global x
           x = 300

         myfunc()

         print(x)
```

```
300
```

Also, use the global keyword if you want to make a change to a global variable inside a function.

**Example:**

To change the value of a global variable inside a function, refer to the variable by using the global keyword:

```
In [30]: x = 300

         def myfunc():
           global x
           x = 200

         myfunc()

         print(x)
```

```
200
```

## 2.7 Recursive Function:

A recursive function is a function that calls itself in order to solve a problem. Recursion is a powerful tool in programming and can be used to simplify solutions to complex problems. However, it requires careful handling to avoid infinite loops and stack overflow errors.

1. **Base Case:** This is the condition under which the function stops calling itself. Without a base case, the function would call itself indefinitely.
2. **Recursive Case:** This is the part of the function where it calls itself with a smaller or simpler input.

A function that calls itself is known as **Recursive Function**.

**Example:** factorial(3) =3*factorial(2)

=3*2*factorial(1)

=3*2*1*factorial(0)

=3*2*1*1 =6

factorial(n)= n*factorial(n-1)

**The main advantages of recursive functions are:**

1. We can reduce length of the code and improves readability.
2. We can solve complex problems very easily. For example, Towers of Hanoi, Ackerman's Problem etc.,

**Example1:**

Write a Python Function to find factorial of given number with recursion.

```
In [1]: def factorial(n):
            if n==0:
                result=1
            else:
                result=n*factorial(n-1)
            return result
        print("Factorial of 0 is :",factorial(0))
        print("Factorial of 4 is :",factorial(4))
        print("Factorial of 5 is :",factorial(5))
        print("Factorial of 40 is :",factorial(40))
```

```
Factorial of 0 is : 1
Factorial of 4 is : 24
Factorial of 5 is : 120
Factorial of 40 is : 815915283247897734345611269596115894272000000000
```

**Example2:**

**Program:** Write a Python Function to find the fibonacci series of given number with recursion.

```
def fibonacci(n):
    # Base cases
    if n == 0:
        return 0
    elif n == 1:
        return 1
    # Recursive case
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

# Example usage
print(fibonacci(8))  # Output: 8
```

```
21
```

## 2.8 Recursion vs Iteration:

Recursion and iteration are two fundamental programming concepts used to solve repetitive problems. Both can often be used to solve the same problem, but they do so in different ways.

1. **Recursion:**
   - Recursion is a method where a function calls itself in order to solve a problem. It breaks down a problem into smaller sub-problems until it reaches a base case, which stops the recursion.
   - **Example:** Factorial Calculation Using Recursion

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(5))  # Output: 120
```

```
120
```

   **Advantages of Recursion:**
   - **Simpler Code:** Recursive solutions can be more concise and easier to understand, particularly for problems that naturally fit a recursive pattern (e.g., tree traversals).
   - **Intuitive:** It can be more intuitive for certain problems, like those defined by recurrence relations.

   **Disadvantages of Recursion:**
   - **Performance:** Recursion can be less efficient due to the overhead of multiple function calls and maintaining a call stack.
   - **Memory Usage:** Each function call adds a new frame to the call stack, which can lead to stack overflow if the recursion depth is too large.
   - **Complexity:** It can be more difficult to debug and understand, especially for those not familiar with the concept.
2. **Iteration:**
   - Iteration uses loops (like for and while) to repeat a block of code until a condition is met.
   - **Example:** Factorial Calculation Using Iteration

```
def factorial_iterative(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

print(factorial_iterative(5))  # Output: 120
```

```
120
```

**Advantages of Iteration:**

- **Efficiency:** Iterative solutions often have lower overhead and are more efficient in terms of both time and space.
- **Memory Usage:** Iteration does not have the same issues with call stack size, making it safer for very deep or infinite loops.
- **Simplicity:** Easier to understand and debug for those familiar with basic loop constructs.

**Disadvantages of Iteration:**

- **Complexity of Code:** Iterative solutions can sometimes be more verbose and harder to write correctly, especially for problems that naturally fit a recursive definition.
- **Less Intuitive:** It may be less intuitive for problems that are inherently recursive, like tree traversals or mathematical sequences.

**Comparison Table**

| Feature | Recursion | Iteration |
|---|---|---|
| Definition | Function calls itself | Uses loops to repeat code |
| Simplicity | Can be simpler and more readable | Can be verbose but straightforward |
| Efficiency | May have higher overhead | Generally, more efficient |
| Memory | Uses call stack, may cause overflow | Does not risk stack overflow |
| Use cases | Naturally recursive problems | Problems with definite loops |
| Debugging | Can be harder to debug | Easier to debug and trace |

## Assignment Questions:

### Conditional Statements

1. Explain the importance of conditional statements in Python programming. Provide examples to demonstrate the usage of if, elif, and else statements.
2. Describe how nested conditional statements work in Python. Write a program that uses nested if statements to determine the grade of a student based on their score.

### Loops

3. Compare and contrast the while loop and the for loop in Python. Provide examples to illustrate their use cases and scenarios where one might be preferred over the other.
4. Explain the range() function in Python. How is it used in conjunction with loops? Write a program that prints all the even numbers between 1 and 50 using the range() function and a for loop.

### Nested Loops

5. Describe the concept of nested loops in Python. Write a program that uses nested loops to print a multiplication table for numbers 1 through 10.
6. Discuss the potential issues with nested loops, such as complexity and performance. Provide examples of situations where nested loops might lead to inefficient code and suggest possible alternatives.

### Loop Control Statements

7. Explain the purpose of the break, continue, and pass statements in Python. Provide examples to demonstrate their usage within loops.
8. Describe the while-else and for-else constructs in Python. How do they differ from regular loops? Provide examples to illustrate their functionality.

### Functions

9. Define a function in Python and explain its syntax. Write a simple function that calculates the factorial of a given number and demonstrate how to call this function.
10. Discuss the different types of arguments that can be passed to a function in Python, including positional, keyword, default, and variable-length arguments. Provide examples for each type.

### Scope of Variables

11. Explain the concept of local and global scope in Python. How does Python handle variable scope within functions? Provide examples to illustrate the differences between local and global variables.

12. Write a program that demonstrates the use of global variables within a function. Explain the global keyword and its significance in managing variable scope.

**Return Statement**

13. Discuss the return statement in Python functions. What is its purpose, and how does it affect the function's output? Write a function that returns multiple values and demonstrate how to capture them.

14. Explain the difference between returning None and returning a value from a function. Provide examples to illustrate scenarios where each approach is appropriate.

**Recursion**

15. Define recursion and explain how it works in Python. Write a recursive function to calculate the Fibonacci sequence up to a given number. Compare the efficiency of recursion with iteration for this problem.

16. Discuss the pros and cons of using recursion versus iteration in Python. Provide examples where recursion is more suitable than iteration and vice versa. Include considerations such as readability, performance, and stack depth limitations.