Control Flow in Python:

Frequently, a program needs to skip over some statements, execute a series of statements repetitively, or choose between alternate sets of statements to execute.

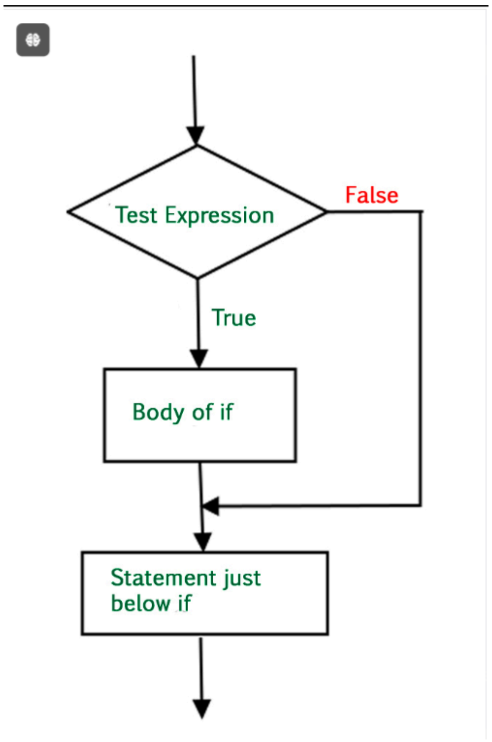
That is where control structures come in. A control structure directs the order of execution of the statements in a program (referred to as the program's control flow).

1. Conditional Statements

i. if statement

The if statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not.

Flowchart:



Example:

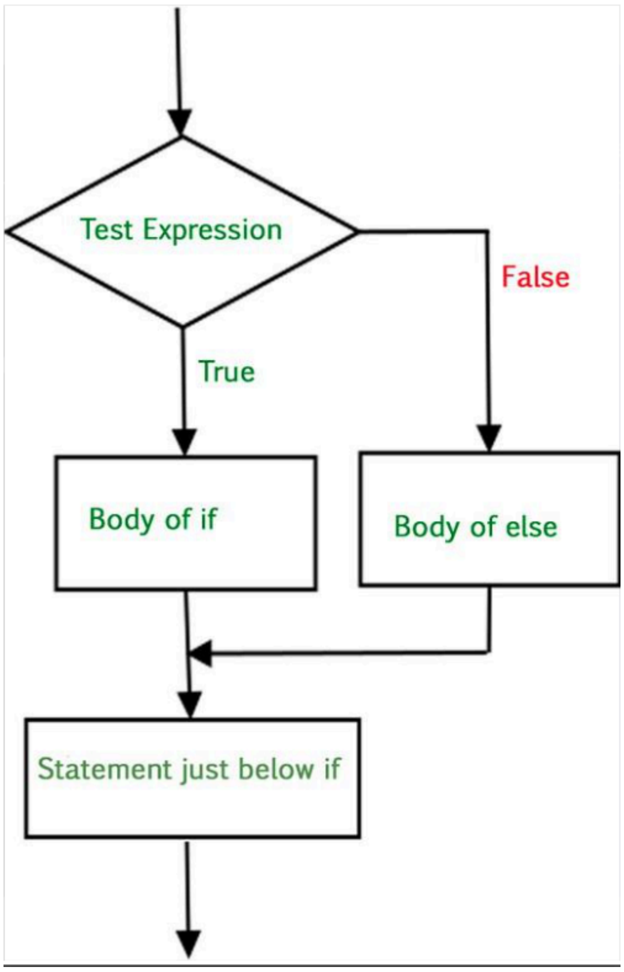
```
age = 21
if age > 18:
    print("You are eligible to vote")
print("You are nepali citizen")
```

Output:

```
You are nepali citizen
```

ii. If-else statement

The if statement in Python allows you to execute a block of code when a condition is true. However, if we want to specify an alternative action when the condition is false, we can pair the if statement with an else clause. This way, when the condition is false, the block of code within the else statement will be executed.
FLowchart:



Example:

```
age = 21
if age > 18:
    print("You are eligible to vote")
else:
    print("You are not eligible to vote")
print("You are nepali citizen")
```

Output:

```
You are eligible to vote
You are nepali citizen
```

iii. if-elif-else statement

In Python, the if, elif, and else statements are used to control the flow of execution based on multiple conditions.

- The if statement checks the first condition.
- The elif (short for "else if") statement checks additional conditions if the previous conditions are false.
- The else statement is executed if none of the previous conditions are true.

Example:

```
temperature = 30

if temperature > 35:
    print("It's a hot day.")
elif temperature > 20:
    print("It's a warm day.")
elif temperature > 10:
    print("It's a cool day.")
else:
    print("It's a cold day.")
```

⇒ It's a warm day.

iv. if-elif statement

The if and elif statements allow us to check multiple conditions without necessarily needing an else. Each if and elif checks a condition, and if one is true, the corresponding block of code will execute. If no conditions are met, nothing happens—there is no fallback like in an else block.

Example:

```
age = 18

if age >= 21:
    print("You can legally drink.")
elif age >= 18:
    print("You can vote.")
elif age >= 16:
    print("You can drive.")
```

⇒ You can vote.

Switch statement in Python

Before Python 3.10, Python developers had to use multiple if-elif-else statements or dictionaries to simulate switch case functionality. With Python 3.10, the match statement and case keywords were introduced, providing a more elegant solution similar to traditional switch case statements in other languages.

Example:

Previously:

```
day = input("Enter the day of the week: ")

if day == "Saturday" or day == "Sunday":
    print(f"{day} is a weekend.")
elif   day   in   ["Monday",   "Tuesday",   "Wednesday",   "Thursday",
"Friday"]:
    print(f"{day} is a weekday.")
else:
    print("That's not a valid day of the week.")
```

After Python 3.10

```
day = input("Enter the day of the week: ")

match day:
    case "Saturday" | "Sunday":
        print(f"{day} is a weekend.")
       case  "Monday"  |  "Tuesday"  |  "Wednesday"  |  "Thursday"  |
"Friday":
        print(f"{day} is a weekday.")
    case _:
        print("That's not a valid day of the week.")
```

2. Iterative Statements

The iterative statements are also known as looping statements or repetitive statements. The iterative statements are used to execute a part of the program repeatedly as long as a given condition is True. Python provides the following iterative statements.

- while statement
- for statement

There is no do-while loop in python.

i. for loop

Syntax:

```
For x in sequence:
        Body
```

x ⇒ temporary variable
Sequences are list, set, tuple, dictionary, string, range.

Example:

```
s = input("Enter any string: ")
i = 0
for x in s:
    print(f"The character present at {i} index is {x}")
    i = i + 1
```

⇒

```
Enter any string: nepal
```

```
The character present at 0 index is n
The character present at 1 index is e
The character present at 2 index is p
The character present at 3 index is a
The character present at 4 index is l
```

Program to display table of 8 using for loop:

```
number = 8

for i in range(1, 11):
    print(f"{number} x {i} = {number * i}")
```

⇒

```
8 x 1 = 8
8 x 2 = 16
8 x 3 = 24
8 x 4 = 32
8 x 5 = 40
8 x 6 = 48
8 x 7 = 56
8 x 8 = 64
8 x 9 = 72
8 x 10 = 80
```

Using List

```
list = eval(input("Enter a list: "))
sum = 0
for x in list:
    sum = sum + x

print("The sum is ", sum)
```

Output:

```
Enter a list: [100,200,300,400]
The sum is  1000
```

ii. While Loop

The while statement is used to execute a set of statements repeatedly. In Python, the while statement is also known as entry control loop statement because in the case of the while statement, first, the given condition is verified then the execution of statements is determined based on the condition result.

The general syntax of the **while** statement in Python is as follows.

```
⇒ while condition:
        statement_1
        statement_2
        …
```

Example:

```
count = int(input('How many times you want to say "Hello": '))
i = 1
while i <= count:
    print('Hello')
    i += 1
print('Job is done! Thank you!!')
```

Output:

```
How many times you want to say "Hello": 5
Hello
Hello
Hello
Hello
Hello
Job is done! Thank you!!
```

# To print sum of first n natural numbers

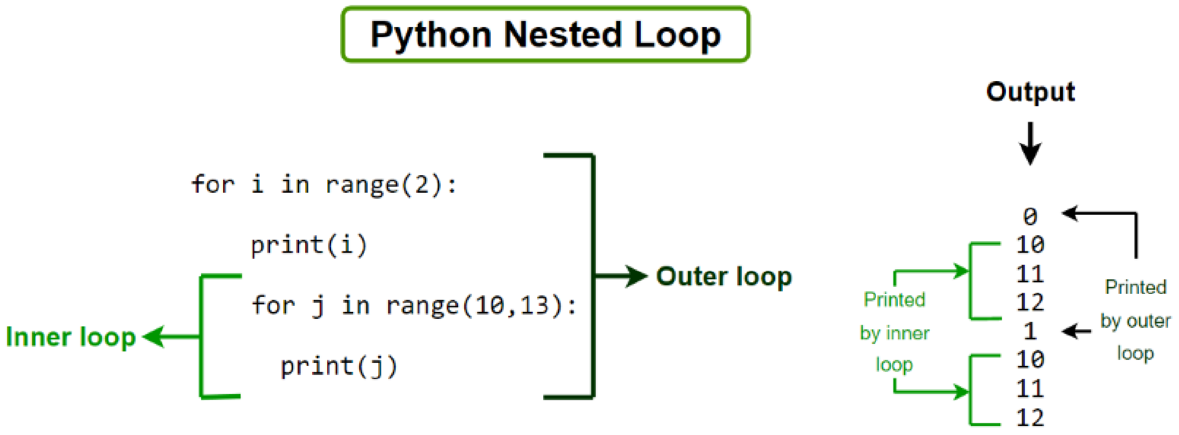
```
n = int(input("Enter the value of n: "))
sum = 0
i = 1
while i <= n:
    sum = sum + i
    i += 1
print(f"The sum of first {n} number is ", sum)
```

⇒

```
Enter the value of n: 5
The sum of first 5 number is  15
```

Nested loop in python

We can use both for loop and while loop to create a nested loop. Nested loops mean loops inside a loop. For example, while loop inside the for loop, for loop inside the for loop, etc.



```
for i in range(2):
    print(i)
    for j in range(10,13):
        print(j)
```

```
0
10
11
12
1
10
11
12
```

Example:

```
for i in range(3):
    for j in range(2):
        print("Hello World")
```

Can you guess how many times Hello World will be printed?

⇒

```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

Nested Loops are mostly used in printing pattern:

Example:

```
n = int(input("Enter the number of rows: "))
for i  in range(n):
    print("@"*4)
```

Output:

```
@@@@
@@@@
@@@@
@@@@
```

Transfer Statements in Python

Using loops in Python allows us to automate and repeat tasks efficiently. However, there may be situations where we want to exit a loop completely, skip an iteration, or ignore a specific condition. This can be achieved using loop control statements, which change the normal execution sequence of the loop. When execution exits a scope, all automatic objects created in that scope are destroyed. Python supports the following control statements:

- Break Statement
- Continue Statement
- Pass Statement

Break Statement:

The break statement is used to terminate the loop or statement in which it is present. Once a break statement is executed, control passes to the statements following the break statement, if any. If the break statement is inside a nested loop, it only terminates the innermost loop containing the break.

Example -01 :

```
while True:
    user_input = input("Enter a number: ")

    # Check if the user wants to exit
    if user_input == '0':
        break  # Exit the loop if the input is 0

    print(f"You entered: {user_input}")

print("Exited the loop.")
```

⇒

```
Enter numbers to continue. Enter 0 to exit.
Enter a number: 5
You entered: 5
Enter a number: 10
You entered: 10
Enter a number: 0
Exited the loop.
```

Continue Statement

The continue statement is another loop control statement. Unlike break, it forces the loop to skip the current iteration and proceed to the next iteration. When a continue statement is executed, any code following it in the loop for that iteration is skipped.

Example:

Suppose you are in the supermarket, and you are buying different things but you cannot buy any item which is priced above 500, so you continue to move without adding it to the cart.
This can be illustrated using the continue statement.
Example:

```
items_price = [10, 20, 80, 450, 360, 550, 30, 90, 100,1000,75]
for i in items_price:
    if i >= 500:
        print("You cannot buy this item")
        continue
    print(i)
print("Shopping Completed")
```

⇒

```
10
20
80
450
360
You cannot buy this item
30
90
100
You cannot buy this item
75
Shopping Completed
```

Pass Statement

The pass statement serves as a placeholder in Python. It does nothing when executed and is used where a statement is syntactically required but no action is desired. It can be useful for writing empty loops, functions, and classes. It can also be used when user doesn't know what to implement now, but can be added later.

```
s = "Nepal"

# Empty loop
for i in s:
    # No error will be raised
    pass

# Empty function
def fun():
    pass

# No error will be raised
fun()

# Pass statement in a loop
for i in s:
    if i == 'k':
        print('Pass executed')
        pass
    print(i)
```

⇒

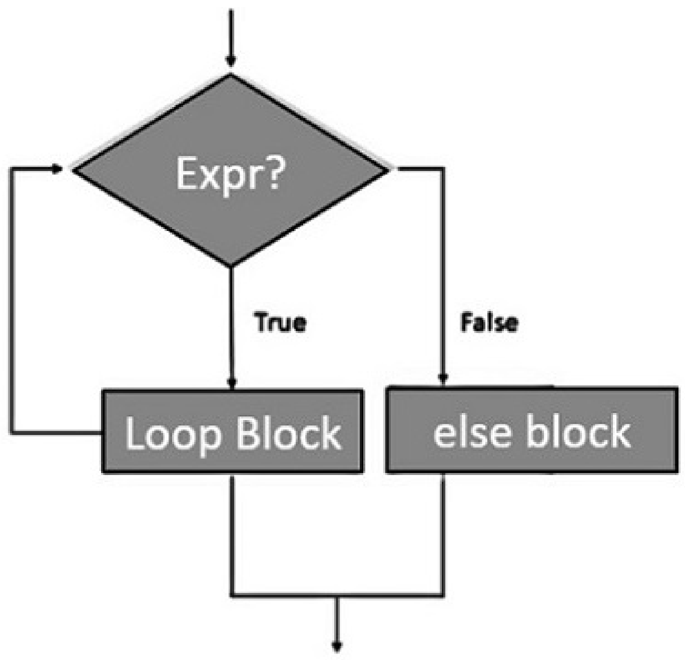
```
N
e
p
a
l
```

for - else loop

Python supports an optional else block to be associated with a for loop. If an else block is used with a for loop, it is executed only when the for loop terminates normally.

The for loop terminates normally when it completes all its iterations without encountering a break statement, which allows us to exit the loop when a certain condition is met.

Flowchart:



Example:

Finding a prime number:

```python
number = int(input("Enter a number to check if it is prime: "))

if number < 2:
    print(f"{number} is not a prime number.")
```

```
else:
    for i in range(2, number):
        if number % i == 0:
            print(f"{number} is not a prime number.")
            break
    else:
        print(f"{number} is a prime number.")
```

Output:

```
Enter a number to check if it is prime: 9
9 is not a prime number.


Enter a number to check if it is prime: 7
7 is a prime number.
```

del statement

The del statement in Python is used to delete objects or variables. It can remove items from a list, delete variables, or even delete entire objects. Here's a brief overview along with examples:

Syntax:

```
del <variable_name>  # Deletes a variable
del <list>[<index>]  # Deletes an item from a list at a specified
index
del <dictionary>[<key>]  # Deletes a key-value pair from a
dictionary
```

Example:

a = 10
print(a)
del a
print(a)

Output:

```
10
Traceback (most recent call last):
  File "g:\INTERN\05-Day5\iterative\del.py", line 4, in <module>
    print(a)
          ^
NameError: name 'a' is not defined
```