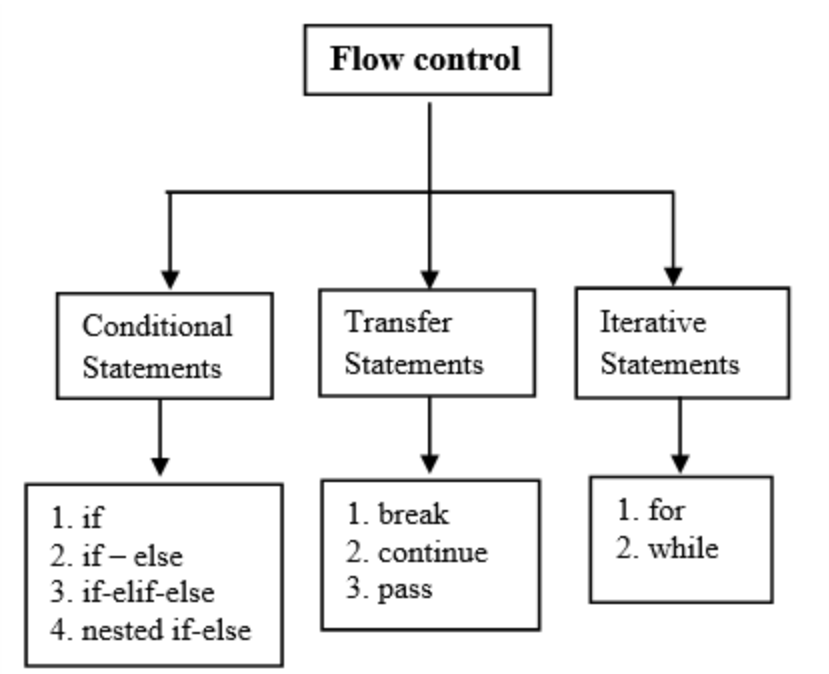
Python Control Flow Statements and Loops

In Python programming, flow control is the order in which statements or blocks of code are executed at runtime based on a condition.

Control Flow Statements

The flow control statements are divided into **three categories**

1. Conditional statements
2. Iterative statements.
3. Transfer statements



Python control flow statements

Conditional statements

In Python, condition statements act depending on whether a given condition is true or false. You can execute different blocks of codes depending on the

outcome of a condition. Condition statements always evaluate to either True or False.

There are three types of conditional statements.

1. if statement
2. if-else
3. if-elif-else
4. nested if-else

Iterative statements

In Python, iterative statements allow us to execute a block of code repeatedly as long as the condition is True. We also call it a loop statements.

Python provides us the following two loop statement to perform some actions repeatedly

1. for loop
2. while loop

Let's learn each one of them with the examples

Transfer statements

In Python, transfer statements are used to alter the program's way of execution in a certain manner. For this purpose, we use three types of transfer statements.

1. break statement
2. continue statement
3. `pass` statements

If statement in Python

In control statements, The `if` statement is the simplest form. It takes a condition and evaluates to either `True` or `False`.

If the condition is `True`, then the True block of code will be executed, and if the condition is False, then the block of code is skipped, and The controller moves to the next line

Syntax of the `if` statement

```
if condition:
     statement 1
     statement 2
     statement n
```

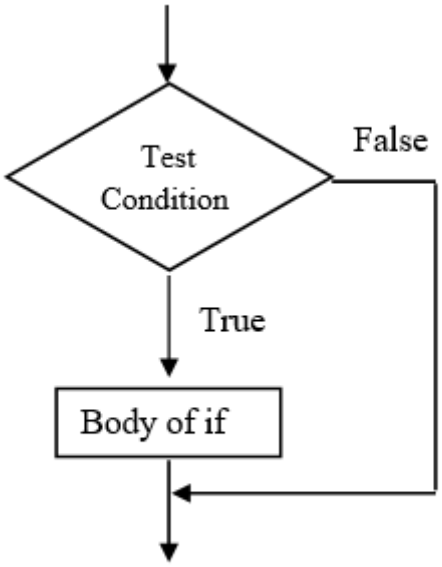


Fig. Flowchart of if statement

Python if statements

Let's see the example of the if statement. In this example, we will calculate the square of a number if it greater than 5

Example

```
number = 6
if number > 5:
    # Calculate square
    print(number * number)
```

```
print('Next lines of code')
```

Output

```
36

Next lines of code
```

If – else statement

The `if-else` statement checks the condition and executes the `if` block of code when the condition is True, and if the condition is False, it will execute the `else` block of code.

Syntax of the `if-else` statement

```
if condition:
    statement 1
else:
    statement 2
```

If the condition is `True`, then statement 1 will be executed If the condition is `False`, statement 2 will be executed. See the following flowchart for more detail.

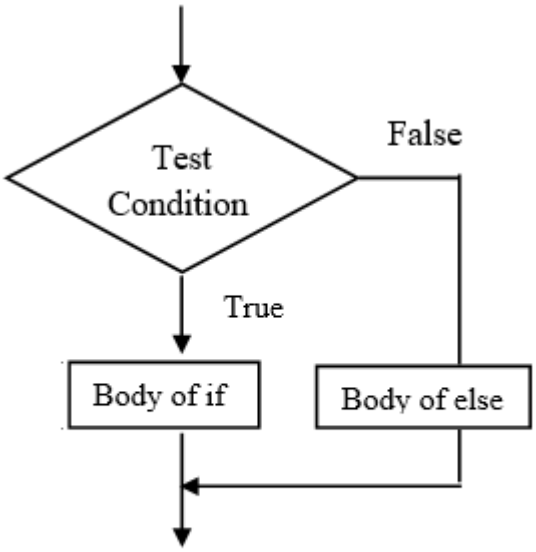


Fig. Flowchart of if-else

Python if-else statements

Example

```
password = input('Enter password ')

if password == "PYnative@#29":
    print("Correct password")
else:
    print("Incorrect Password")
```

Output 1:

```
Enter password PYnative@#29

Correct password
```

Output 2:

```
Enter password PYnative

Incorrect Password
```

Chain multiple if statement in Python

In Python, the `if-elif-else` condition statement has an `elif` blocks to chain multiple conditions one after another. This is useful when you need to check multiple conditions.

With the help of `if-elif-else` we can make a tricky decision. The `elif` statement checks multiple conditions one by one and if the condition fulfills, then executes that code.

Syntax of the `if-elif-else` statement:

```
if condition-1:
     statement 1
elif condition-2:
     stetement 2
elif condition-3:
     stetement 3
     ...
else:
     statement
```

Example

```
def user_check(choice):
    if choice == 1:
        print("Admin")
    elif choice == 2:
        print("Editor")
    elif choice == 3:
        print("Guest")
    else:
        print("Wrong entry")

user_check(1)
user_check(2)
user_check(3)
user_check(4)
```


Output:

```
Admin

Editor
```

```
Guest

Wrong entry
```

Nested if-else statement

In Python, the nested `if-else` statement is an if statement inside another `if-else` statement. It is allowed in Python to put any number of `if` statements in another `if` statement.

Indentation is the only way to differentiate the level of nesting. The nested `if-else` is useful when we want to make a series of decisions.

Syntax of the nested-`if-else`:

```
if conditon_outer:
    if condition_inner:
        statement of inner if
    else:
        statement of inner else:
    statement ot outer if
else:
    Outer else
statement outside if block
```

Example: Find a greater number between two numbers


```
num1 = int(input('Enter first number '))
num2 = int(input('Enter second number '))

if num1 >= num2:
    if num1 == num2:
        print(num1, 'and', num2, 'are equal')
    else:
        print(num1, 'is greater than', num2)
else:
    print(num1, 'is smaller than', num2)
```

Output 1:

```
Enter first number 56

Enter second number 15
```

```
56 is greater than 15
```

Output 2:

```
Enter first number 29

Enter second number 78

29 is smaller than 78
```

Single statement suites

Whenever we write a block of code with multiple if statements, indentation plays an important role. But sometimes, there is a situation where the block contains only a single line statement.

Instead of writing a block after the colon, we can write a statement immediately after the colon.

Example

```
number = 56
if number > 0: print("positive")
else: print("negative")
```

Similar to the `if` statement, while loop also consists of a single statement, we can place that statement on the same line.

Example

```
x = 1
while x <= 5: print(x,end=" "); x = x+1
```

Output

```
1 2 3 4 5
```

for loop in Python

Using for loop, we can iterate any sequence or iterable variable. The sequence can be string, list, dictionary, set, or tuple.

Read the Complete guide on **Python for loop**.

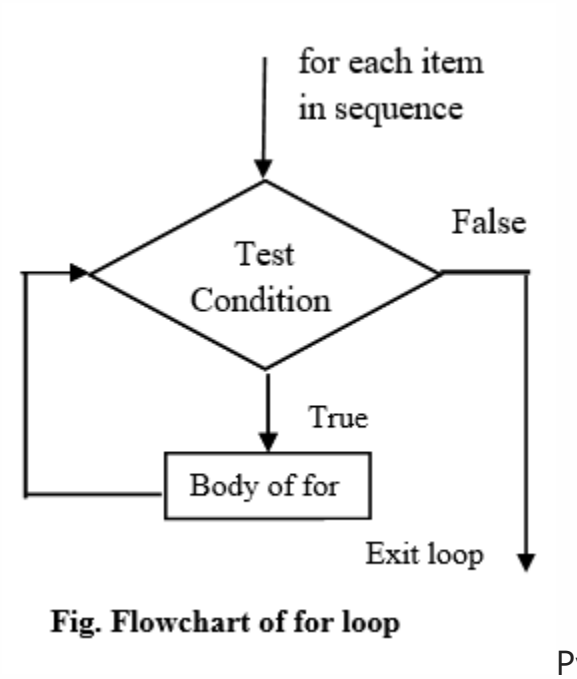


Fig. Flowchart of for loop

Python for loop

Syntax of `for` loop:

```
for element in sequence:
     body of for loop
```

Example to display first ten numbers using for loop

```
for i in range(1, 11):
    print(i)
```

Output

```
1

2
```

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

Also, read **Nested loops in Python**.

While loop in Python

In Python, The while loop statement repeatedly executes a code block while a particular condition is true.

In a while-loop, every time the condition is checked at the beginning of the loop, and if it is true, then the loop's body gets executed. When the condition became False, the controller comes out of the block.

Read the Complete guide on **Python while loop**.

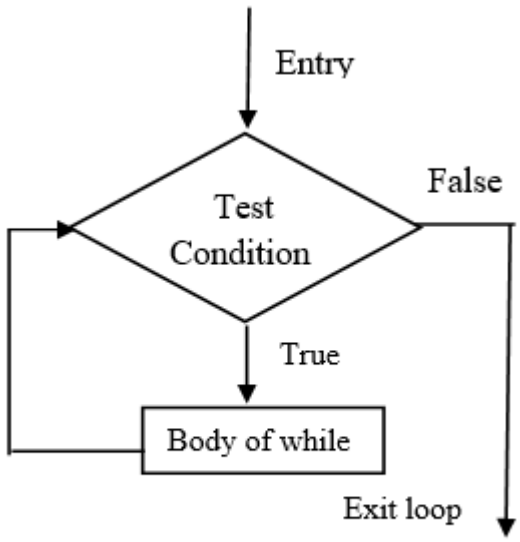


Fig. Flowchart of while loop

Python while loop

Syntax of while-loop

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

Example to calculate the sum of first ten numbers

```
num = 10
sum = 0
i = 1
while i <= num:
    sum = sum + i
    i = i + 1
print("Sum of first 10 number is:", sum)
```


Output

```
Sum of first 10 number is: 55
```

Break Statement in Python

Read: Complete guide on Python Break, Continue, and Pass.

The break statement is used inside the loop to exit out of the loop. It is useful when we want to terminate the loop as soon as the condition is fulfilled instead of doing the remaining iterations. It reduces execution time. Whenever the controller encountered a break statement, it comes out of that loop immediately

Let's see how to break a for a loop when we found a number greater than 5.

Example of using a break statement

```
for num in range(10):
    if num > 5:
        print("stop processing.")
        break
    print(num)
```

Output

```
0

1

2

3

4

5

stop processing.
```


Continue statement in python

The continue statement is used to skip the current iteration and `continue` with the next iteration.

Let's see how to skip a for a loop iteration if the number is 5 and continue executing the body of the loop for other numbers.

Example of a `continue` statement

```
for num in range(3, 8):
    if num == 5:
        continue
    else:
        print(num)
```

Output

```
3

4

6

7
```

Pass statement in Python

The pass is the keyword In Python, which won't do anything. Sometimes there is a situation in programming where we need to define a syntactically empty block. We can define that block with the pass keyword.

A pass statement is a Python null statement. When the interpreter finds a pass statement in the program, it returns no operation. Nothing happens when the `pass` statement is executed.

It is useful in a situation where we are implementing new methods or also in exception handling. It plays a role like a placeholder.

Example

```
months = ['January', 'June', 'March', 'April']
for mon in months:
    pass
print(months)
```