Section-1: Decision Structures, Boolean Logic, and Control Statements: if, if-else, if-elif-else statements, Nested Decision Structures, Comparing Strings, Logical Operators, Boolean Variables. P1-20
Section-2: Repetition Structures: Introduction, while loop, for loop, Calculating a Running Total, Input Validation Loops, Nested Loops.
Section-3: Strings: Accessing characters and Substring in Strings, Data Encryption, Strings and Number Systems.

Decision Structures

A **decision structure** is a set of program statements that makes a decision and changes the flow of the program based on that decision. These are also called **Selection or Control Flow Statements**. The decisions are made based on **True** or **False** of a **Boolean Logic test.**

The **control flow statements** are classified as follows:

A. **Selection or Decision or Conditional Statements**
    a. **if, if-else, elif**
B. **Loop or Repetition or Iterative Statements**
    a. **for, while**
C. **Jump Statements**
    a. **break, continue, pass**

A. **Selection or Decision or Conditional Statements**

In decision statements, the conditional expressions are evaluated with an outcome of either True or False.

a. The selection statements are 3 types:

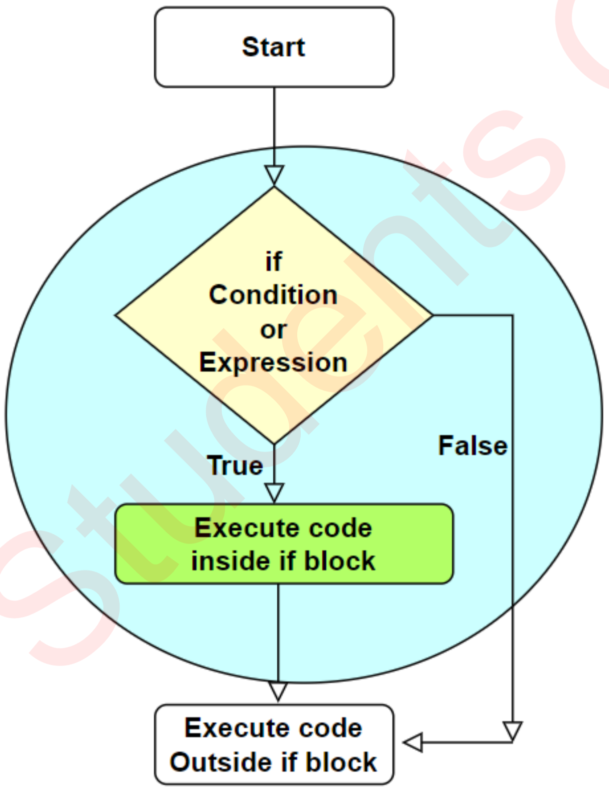
Type	**Single Selection**	**Two-Way Selection**	**Multi-Way Selection**
Command	**if** statement	**if - else** statement	Nested **if - else** statements **elif** Ladder statements

Single Selection in Python ("if" statement)

- "if" is a simple selection statement in Python. It is used to modify the flow of execution of a program.
- "if" consists of a **condition (boolean expression), colon :, and a block of statements with the same indentation**,
    - When the condition is **True**, the **'if' block of statements** will be executed,
    - Otherwise, the first statement outside the 'if' block will be executed.
- All the statements inside the 'if' block must have the same indentation of spaces

Syntax:

```
if condition:
    True block of statements
    statement 1
    statement 2
    ...
    statement n

Statements outside if block
```


Start
if Condition or Expression
True
False
Execute code inside if block
Execute code Outside if block

Application:

```
#if statement example
m, n = 77, 87
if(m < n):
    result = "m is smaller than n"
    print(result)
```

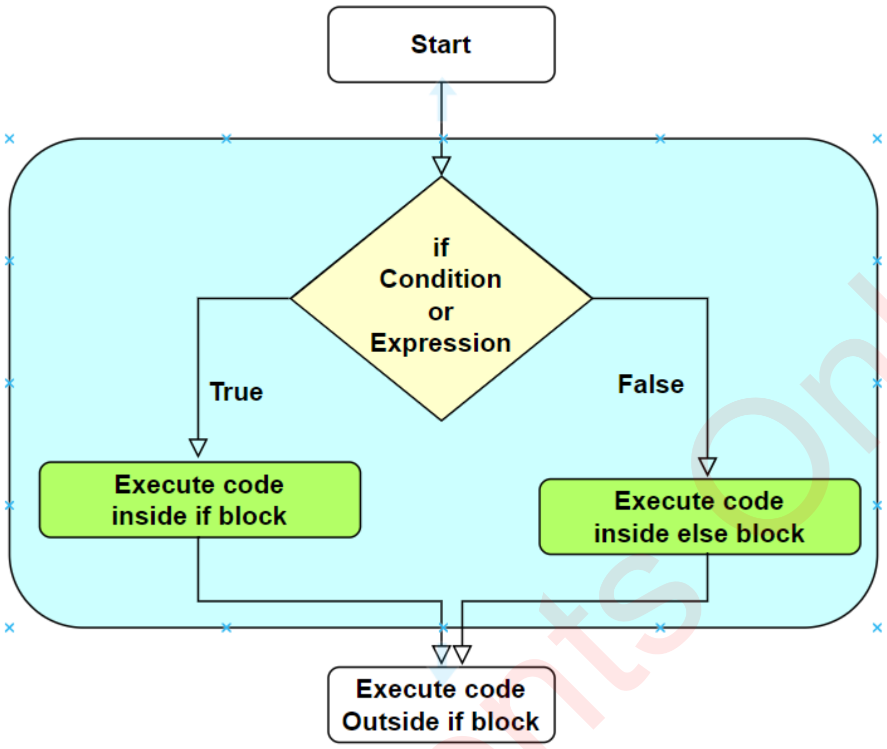
```
Output:
m is smaller than n
```

Two-Way Selection in Python ("if-else" statement)

- An **if** statement can also be followed by an optional **else** statement. **if-else** is a two-way decision statement which means, we have only two alternative choices.
- **if** statement will have a Boolean_Expression; **else** statement has NO Boolean_Expression.
- **if-else** consists of a condition (Boolean_Expression), a block of statements for '**if**', and another block of statements for '**else**'.
    - When the Boolean_Expression is **True**, the '**if**' block of statements will be executed
    - When the Boolean_Expression is **False**, the '**else**' block of statements will be executed
- **Indentation:**
    - All the statements inside the '**if**' block **must have the same indentation** with spaces
    - All the statements inside the '**else'** block **must have the same indentation** with spaces
    - **However, a different indentation can be used for 'if' block and 'else' block.**

Syntax:

```
if condition:
    True block of statements
    statement 1
    statement 2
    ...
    statement n
else:
    False block of statements
    statement 1
    statement 2
    ...
    statement n

Statements outside if-else block
```


Start
if Condition or Expression
True
False
Execute code inside if block
Execute code inside else block
Execute code Outside if block

Application-1:

```python
'''
if-else construct using Membership operator 'in' and 'set' of names
Strings are case sensitive in Python;
Upper case strings are different from Lower case strings
'''
cse = {"Saida", "Ajay", "Sai", "Veda"}
sname = input("Enter a name to search : ")
if sname in cse:
    print("Yes, {} is in CSE branch!".format(sname))
else:
    print("No, {} is Not in CSE branch!".format(sname))
```

Output:

```
Enter a name to search : Veda
Yes, Veda is in CSE branch!

Enter a name to search : veda
No, veda is Not in CSE branch!
```

Application-2:

```
'''
Aim: Program to Check whether the given number is Even or Odd.
'''
num = int(input("Enter an integer : "));
# true if num is perfectly divisible by 2
if(num % 2 == 0):
    print("{} is even.".format(num))
else:
  print("{} is odd.".format(num))
# Notice that if block has 4 space indentation and else block has 2 space indentation
```

Output:

```
Enter an integer : 7
7 is odd.
Enter an integer : 4
4 is even.
```

Application-3:

```
'''
lab-7: Write a program that asks the user for two numbers and prints
Close if the numbers are within .001 of each other and not close
otherwise.
'''
a = float(input("Enter first number : "))
b = float(input("Enter second number : "))
c = abs(a - b)
if c > 0.0009 and c <= 0.001 :
        print("Close")
else :
    print("Not Close")
```

Output:

```
Enter first number : 4.001
Enter second number : 4.002
Close
```

Multi-Way Selection - Nested “if-else”

When a series of decisions is required, the multi-way selection statements are used.
There are 2 types of multi-way selection statements

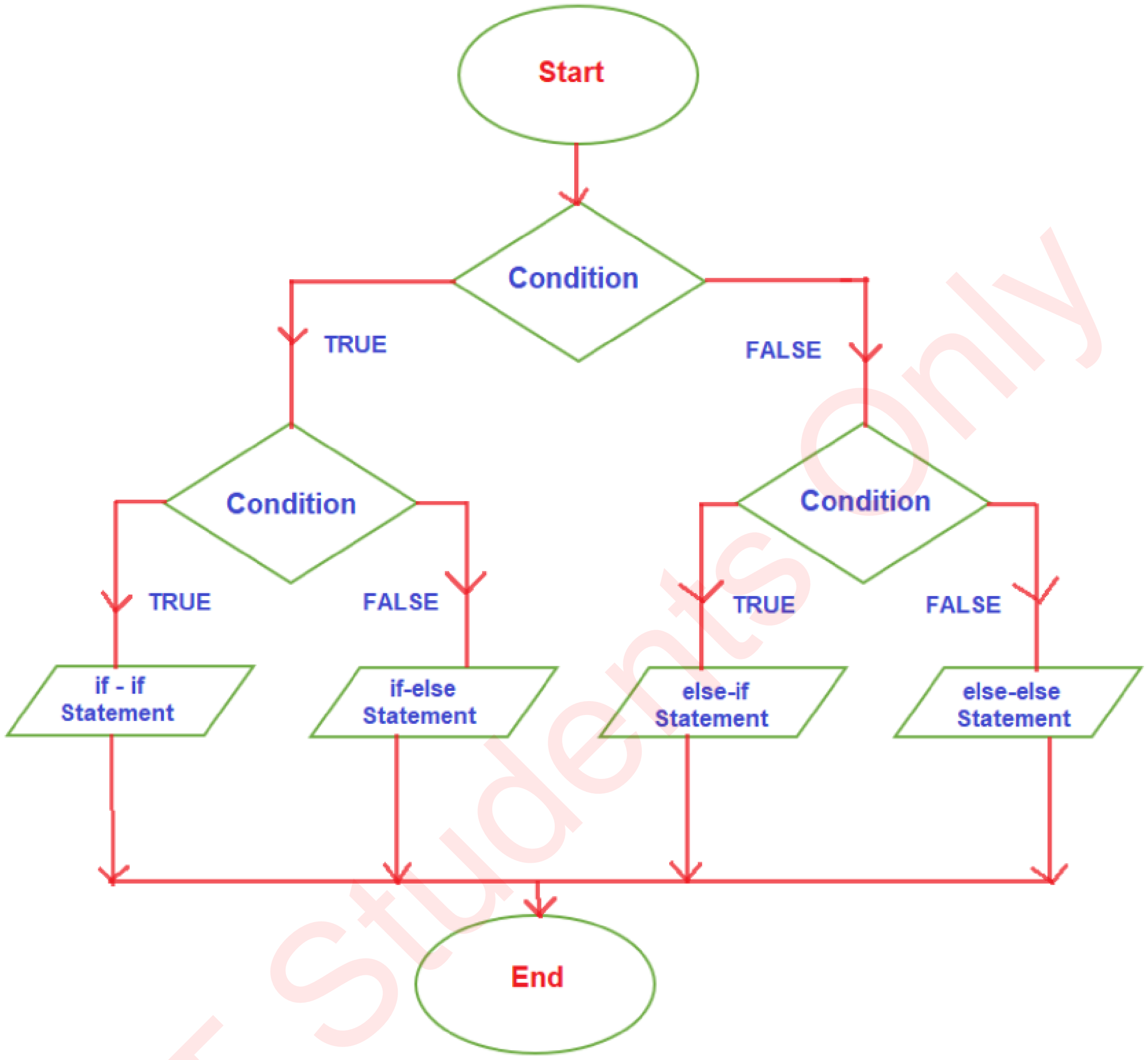
1. **Nested " if-else " statements**
2. **" elif " Ladder statements**

1. **Nested if-else statements**

Nesting means using one "**if-else**" construct within another "**if-else**" construct. Use nested "if-else" when you need to decide more within the parent "if" condition or parent "else" condition. **The nested "if-else" is used when multiple paths of decisions are required.**

Syntax:

```
if(Condition/Expression):         # Outer if block
    if(Condition/Expression):     # Inner if block
        Statements
    else:                         # Inner else block
        Statements
else:                             # Outer else block
    if(Condition/Expression):     # Inner if block
        Statements
    else:                         # Inner else block
        Statements
```



Application-1:

```
#Checks whether input marks are pass or fail
# JustPass=40, pass>40, fail<40
marks = int(input("Enter marks 0-100 : "))
if marks >= 40:
    if marks==40:
        print("Just Passed with ", marks)
    else:
        print("Passed with ", marks);
else:
    print("Failed with ", marks)
```

Output:

```
Enter marks 0-100 : 75
Passed with  75

Enter marks 0-100 : 40
Just Passed with  40

Enter marks 0-100 : 30
Failed with  30
```

Application-2: Write a program to find whether the given year is a Leap Year?

```python
'''
Aim: Find whether the given year is a Leap Year.
Note: A Century year ends with 00 or divisible by 100
Conditions:
Non-Century years that are exactly divisible by 4 are leap years.
A century year divisible by 4, 100, and 400 is a leap year.
A century year divisible by 4, 100, but not divisible by 400 is not a
leap year.

For example,
1900 is not a leap year (Century year, Divisible by 4 & 100; but not
by 400)
1999 is not a leap year (Non-Centuary, Not divisible by 4)
2000 is the leap year   (Century year, Divisible by 4, 100, and 400)
2004 is the leap year   (Non-Centuary, Divisible by 4)
2024 is the leap year   (Non-Centuary, Divisible by 4)
'''
year=int(input('Enter a year:'))
if year%4==0:
    if year%100==0:       # Centuary year
        if year%400==0:   # Century year divisible by 400
            print(f'{year} is a leap year')
        else:             # Century year Not divisible by 400
            print(f'{year} is not leap year')
    else:                 # Non-Centuary year, Divisible by 4
        print(f'{year} is a leap year')
else:                     # not divisible by 4
        print(f'{year} is not a leap year')
```

Output:

```
Enter a year:2022
2022 is not a leap year

Enter a year:2024
2024 is a leap year

Enter a year:3000
3000 is not leap year
```

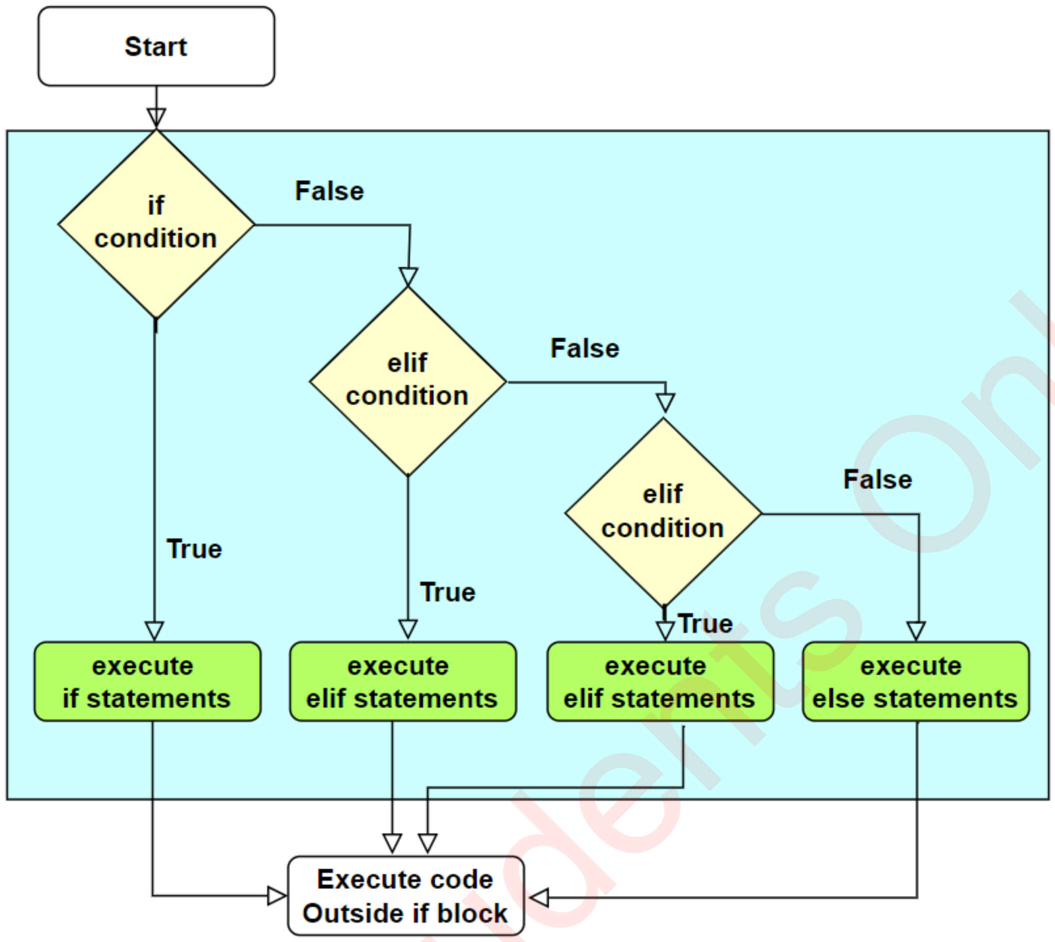
2. “elif” Ladder statements

In Python, “elif” keyword is a short form of “else if”. The “elif” is useful **when you need to decide a series of decisions** after each of the previous “if” conditions.

- The series of conditions are evaluated from **top to bottom**.
- When one condition becomes **true**, the statements of that condition will be executed and the control comes out of the whole “if” block.
- When all the conditions are **false**, then the last default **“else”** statement is executed and the control comes out of the whole “if” block.

Syntax:

```
if boolean_expression1:
    statement(s)
elif boolean_expression2:
    statement(s)
elif boolean_expression3:
    statement(s)
else:
    statement(s)
```



Application-1:

```
# elif conditional statements
x = 20
y = 70
if x > y:
    print("x is greater than y")
elif y > x:
    print ("y is greater than x")
else:
    print("x and y are equal")
```

Output:

```
y is greater than x
```

Application-2: Grades

Write a Program to Prompt for Marks between 0 and 100. If the Marks Is Out of Range, Print an Error. If the Marks are between 0 and 100, Print a Grade Using the Following Table.

Score	>=90	>=80	>=70	>=50	>=40	<40
Grade	A+	A	B	C	D	F

```
# elif conditional statements
marks=int(input("Enter marks 0-100 : "))
if(marks,0 or marks>100):
    print("Marks out of range.")
elif(marks>=90):
    print("Grade A+")
elif(marks>=80):
    print("Grade A")
elif(marks>=70):
    print("Grade B")
elif(marks>=50):
    print("Grade C")
elif(marks>=40):
    print("Grade D")
else:
    print("Grade F")
```

Output:

```
Enter marks 0-100 : 70
Grade B
```

Application-3:

```
'''
Lab-16. Write a program that asks the user to enter a length in feet.
The program should then give the user the option to convert from feet
into inches, yards, miles, millimeters, centimeters, meters, or
kilometers. Say if the user enters a 1, then the program converts to
inches, if they enter a 2, then the program converts to yards, etc.
While this can be done with if statements, it is much shorter with
lists and it is also easier to add new conversions if you use lists.
'''
```

```
feet=int(input("Input distance in feet: "))
print("Choose your option: ")
print("1. inches")
print("2. yards")
print("3. miles")
print("4. millimeters")
print("5. centimeters")
print("6. meters")
print("7. kilometers")
option=int(input("Enter the option : "))
if option==1:
    dist=round(feet*12,2)
    units="inches"
    print("The distance in {} is {} inches.".format(units,dist))
elif option==2:
    dist=feet/3
    units="yards"
    print("The distance in %s is %.2f yards."%(units,dist))
elif option==3:
    #dist=round(feet*0.000189394,3)
    dist=feet/5280
    units="miles"
    print("The distance in %s is %.2f miles."%(units,dist))
elif option==4:
    dist=feet*304.8
    units="millimeters"
    print("The distance in %s is %.2f millimeters."%(units,dist))
elif option==5:
    dist=feet*30.8
    units="centimeters"
    print("The distance in %s is %.2f centimeters."%(units,dist))
elif option==6:
    #dist=round(feet*0.3048,3)
    dist=feet*0.3048
    units="meters"
    print(f"The distance in %s is %.2f meters."%(units,dist))
elif option==7:
    dist=feet/3280.8
    units="kilometers"
    print("The distance in %s is %.2f kilometers."%(units,dist))
else:
    print("Invalid choice!!!")
```

Output:

```
Input distance in feet: 456
Choose your option:
1. inches
2. yards
3. miles
4. millimeters
5. centimeters
6. meters
7. kilometers
Enter the option : 7
The distance in kilometers is 0.14 kilometers.
```

Comparing Strings

In Python, string comparison is the process of comparing two strings to determine whether they are equal or not.

Strings in Python are stored as objects with an ID (memory address).
Python reuses the same memory for two equal strings to save memory, and run faster & easier.

Reference-only

Objects in Python consist of 3 properties:

1. **Identity** - address of the memory where the string is stored
2. **Type** - data type of the string ‘str’
3. **Value** - content stored in the object

Commonly used string comparison methods in Python are,

A. using Built-in **Operators**
B. using Built-in **Functions**

A. <u>Using Built-in Operators for String Comparison:</u>

- Equality/Inequality Operators (**==, !=**) compare similarity
- Identity Operators (**is, is not**) compare address (use **id()** function to find the address of an object)
- Comparision/Relational Operators (**<, <=, >, >=**) compare alphabetical order

Equality/Inequality Operators (==, !=):

The **"=="** and **"!="** operators checks if two strings are equal or not. The strings are case-sensitive. So, upper-case strings are different from lower-case strings.

Syntax: string1 == string2 (returns True if 2 strings are Equal)
Syntax: string1 != string2 (returns True if 2 strings are Not Equal)

Identity Operators (is, is not):

The "**is**" and '**is not**' operators compare the address of two strings and find they are of the same object or different object.
Python considers equal strings as the same object and stores them in the same memory location. So, the '**is**' and '**is not**' operators compare their address locations.

Syntax: string1 is string2
Syntax: string1 is not string2

ASCII	83	**111**	102	116	119	97	114	101
String1	**S**	**o**	**f**	**t**	**w**	**a**	**r**	**e**
String2	**S**	**O**	**F**	**T**	**W**	**A**	**R**	**E**
ASCII	83	**79**	70	84	87	65	82	69

ASCII Values - A-Z : 97-122, a-z : 65-90, 0-9 : 48-57

Application:

```
# Equality/In-Equality operators: ==, !=
# Identity Operators: is, is not   # compare memory address
s1 = "Software"
s2 = "Software"
s3 = "SOFTWARE"
print(s1 == s2)     # True
print(s1 == s3)     # False
print(s1 != s3)     # True

print(s1 is s2)     # True
print(s1 is s3)     # False
print(s1 is not s3) # True
# Notice the address of s1 and s2 are same
print("Address of s1 : ", id(s1))
print("Address of s2 : ", id(s2))
print("Address of s3 : ", id(s3))
```

Output:

```
True
False
True

True
False
True

Address of s1 :  2295811751536
Address of s2 :  2295811751536
Address of s3 :  2295811751344
```

Comparision Operators (<, <=, >, >=) :

The comparison operators check two strings lexicographically, that is based on their alphabetical order. The alphabetical order is determined by comparing the ASCII values of the characters in the strings.

Syntax: string1 > string2
Syntax: string1 < string2

ASCII	67	83	69
String1	C	S	E
String2	A	I	
ASCII	65	73	

Application:

```
dept1 = "CSE"
dept2 = "AI"
if dept1 < dept2:
    print(f"{dept1} comes before {dept2}")
else:
    print(f"{dept2} comes before {dept1}")
```

Output:

```
AI comes before CSE
```

B. Using Built-in Functions for String Comparison:

starstwith() and **endswith()** functions return True or False depending on whether the given substring is found at the beginning, end, or anywhere in the string.
find() function will return the position number (index) of the searched substring in the main string. It returns -1 if the searched string is not found.
count() function will return a number of times the given string has occurred in the main string.

Function	Definition & Syntax	Example s1=”Hi CIT Engineers”

startswith()	Returns True if a string1 starts with a prefix (substring / tuple - True if any one tuple member matches)
	`string.startswith(prefix, start, end)`
		s1.startswith(“Hi”)
		
		t1=(“Hi”, “Hello”)
		s1.startwith(t1)

endswith()	Returns True if a string2 ends with a suffix (substring / tuple - True if any one tuple member matches)
	`string.endswith(suffix,start, end)`
		s1.endswith(“eers”)
		
		t2=(“fine”, “Engineers”)
		s1.endswith(t2)

find()	Returns position# (index#) of substring in string1; Returns -1 if not found. `string.find("substring", start, end)` (optional) start=0, end=last-index	**s1.find(“CIT”)** Output: 3
count()	Returns no.of times given value occurs in a string `string.count("substring", start, end)` (optional) start=0, end=last-index	**s1.count(“i”)** (Default & optional, start=0, end=last)

Application:

```
#String built-in functions startswith(), endswith(), find(), count()
s1 = "Hi CIT Engineers"
if s1.startswith("Hi"):
    print("The string starts with 'Hi'")
tpl=(“Hi”,”Hello”)
result = s1.startswith(tpl):
    print(“Start word in tuple?”,result)   # True
if s1.endswith("eers"):
    print("The string ends with 'Engineers'")
if s1.find("CIT") != -1:
    print("The string contains 'CIT'")
n = s1.count("i")
    print("Number of i letters in the string: ",n)
```

Output:
```
The string starts with 'Hi'
Start word in tuple? True
The string ends with 'Engineers'
The string contains 'CIT'
Number of i letters in the string: 2
```

Logical Operators (and, or, not)

The logical operators are the keywords that **combine multiple conditions into a single condition**. The following table provides information about logical operators.

Operator	Meaning	Example
and	Returns True if all conditions are True otherwise returns False	10 < 5 and 12 > 10 is False
or	Returns False if all conditions are False otherwise returns True	10 < 5 or 12 > 10 is True
not	Returns True if condition is False and returns False if the condition is True	not(10 < 5 and 12 > 10) is True

- **Logical and -** Returns True only if ALL conditions are True, if any one of those conditions is False then whole condition becomes False.
- **Logical or -** Returns True if ANY condition is True, if all conditions are False then the whole condition becomes False.

Application-1: Basic Logical Operators

```
#Logical Opertaors
a = True
b = False
print(a and b) #output: False
print(a or b) #output: True
print(not a) #output: False
a=10
b=5
la = (a<b) and (b<c)    # Combined two conditions
lo = (a<b) or (b<c)     # Combined two conditions
ln = not(a<b)
print("Logical AND = ",la)  #False
print("Logical OR  = ",lo)  #True
print("Logical NOT = ",ln)  #True
```

Application-2: if-else using Logical Operators

```
# Find smallest of three numbers using elif statement
a=10
b=5
c=12
if((a<b) and (a<c)):
    print("a is smallest")
elif ((b<a) and (b<c)):
    print("b is smallest")
else:
    print("c is smallest")
```

Output:

```
b is smallest
```

Boolean Variables

A boolean variable can have only two values: **True** or **False**
The variables with the boolean values **True** or **False** are called Boolean type variables.
These boolean values are case sensitive; hence, the T and F of True and False must be capital letters.

Syntax:

```
Variable = Boolean value
Variable = Boolean expression
```

We can define a boolean variable by simply assigning a True or False value or even an expression that gets evaluated to one of these values.

Application:

```
# Boolean variables & assignment
a = False    # assigned boolean value False
b = True     # assigned boolean value True
print(type(a))
print(type(b))
c = (5>2)    # assigned boolean expression
print("c value : ", c)
print("c data type : ", type(c))
```

Output:

```
<class 'bool'>
<class 'bool'>
c value :  True
c data type :  <class 'bool'>
```

bool() built-in function:

bool() method evaluates any **value or a variable or any expression** and returns a Boolean value either True or False.

bool() method

- returns True for one argument of any value or expression; and
- returns False for `0, None, False, empty values "", (), {}, []`

Syntax:

```
bool()                # False
bool(value)           # True
bool(variable)        # True
bool(expression)      # True
```

Application:

```
# bool() function will always return True for any value
# except 0, None, False, empty values such as "", (), {}, [].
print("Returns True for any value")
print(bool("CSE AI ML"))
print(bool('''Guntur'''))
print(bool(75))
print(bool([10, 20, 40]))
print(bool(-11))
print(bool(3.14))
print(bool(25>(50/3)))
print("Returns False for 0, None, False, empty values \"\", (), {}, []")
print(bool())
print(bool(0))
print(bool(None))
print(bool(False))
print(bool([]))
print(bool(""))
print(bool({}))
print(bool(()))
```

Output:

```
Returns True for any value
True
True
True
True
True
True
True
Returns False for 0, None, False, empty values "", (), {}, []
False
False
False
False
False
False
False
False
```

Section-2: Repetition Structures: Introduction, while loop, for loop, Calculating a Running Total, Input Validation Loops, Nested Loops.

Repetition or Iterative or Looping Structures/Statements

Introduction
In Python, **repetition structures** are used to execute a block of code repeatedly. These structures allow you to execute the same code repeatedly for a finite number of times or until a condition is satisfied.

Each repetition of a block of code is known as a loop or an iteration. So, the repetition structures are also called **looping or iterative structures.**

There are two types of repetition structures in Python..

1. **Indefinite or Condition Controlled Loop -** A loop that repeats an action until the program finds that it needs to stop based on a condition. (**while loop**)
2. **Finite or Sequence Controlled Loop -** A loop that repeats a block of code a predefined number of times or over a sequence of elements. (**for loop**)

Python provides two repetition or looping or iterative statements,

1. **'while' loop**
2. **'for' loop**
3. **Nested loops**

1. **The 'while' loop:**

Definition:

A "**while" loop** executes a block of statements repeatedly until the given condition is **True**.
The "while" loop is used when we DO NOT KNOW the number of iterations. **Entry controlled or a Pre-Test loop** because the '**while' loop** first checks the "condition" to decide if it needs to execute the block of statements. **Event-controlled loop** because the termination of the '**while' loop** depends on an event instead of executing a fixed number of times.

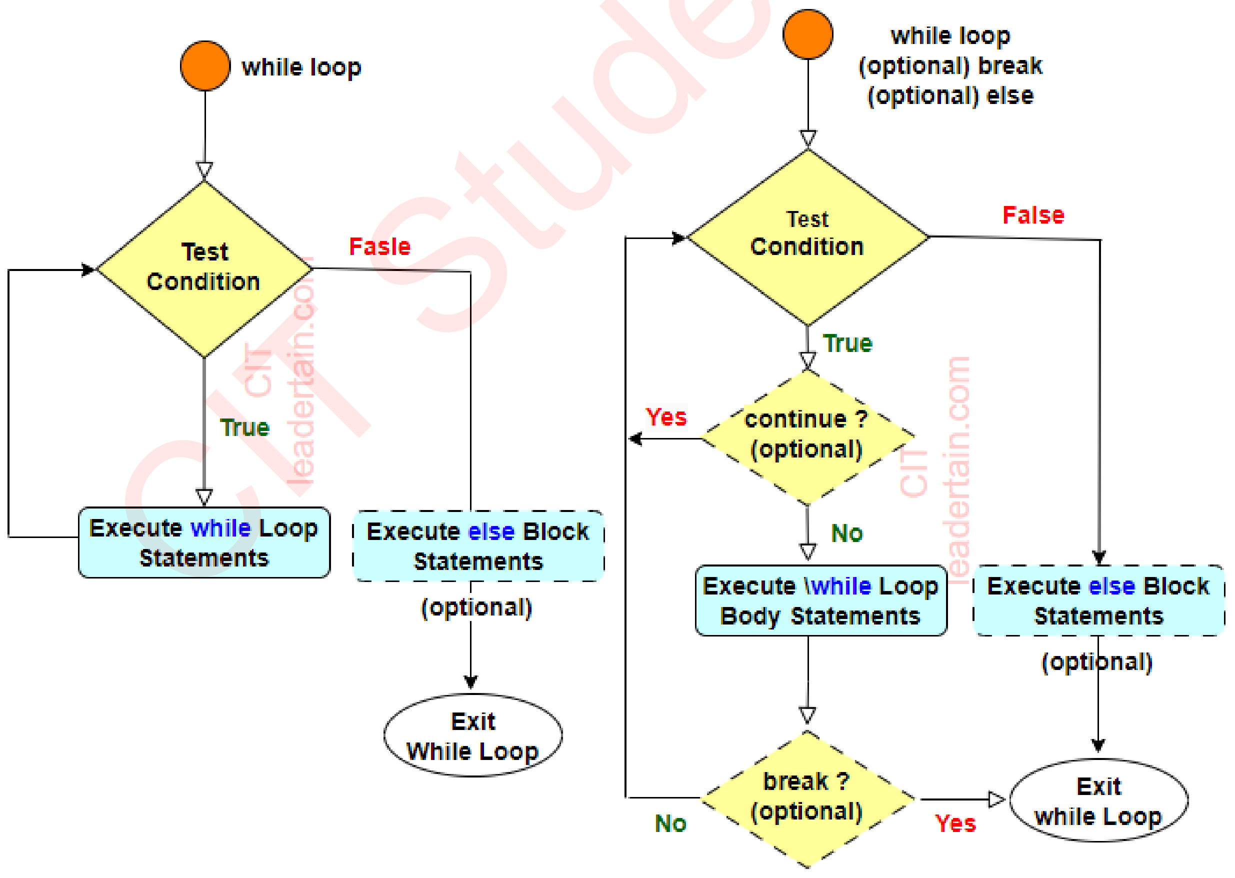
Syntax of 'while" loop:

```
Initialization (optional)
while (condition):
   Loop Body statements
   Incr or Decr (optional)
else:    (optional)
   Block of statements
```
	```
	Initialization (optional)
	while (condition):
	   if (condition):
	         continue (optional)
	   if (condition):
	         break     (optional)
	   Loop Body statements
	   Incr or Decr (optional)
	else:     (optional)
	   Block of statements
	```

Different ways to write 'while' loop condition:

while(true) statements	**while(i<5)** statements	**while(i<=n)** statements

Flow Chart of 'while' loop


while loop
Test Condition
Fasle
True
Execute while Loop Statements
Execute else Block Statements
(optional)
Exit While Loop
while loop (optional) break (optional) else
Test Condition
False
True
continue ? (optional)
Yes
No
Execute \while Loop Body Statements
Execute else Block Statements
(optional)
break ? (optional)
No
Yes
Exit while Loop

WHILE loop:

- Here, the **condition** is a boolean expression that is evaluated before each iteration of the loop. If the condition is **True**, the code inside the loop is executed. This will continue until the condition becomes **False**.

Application: Write a program to print 1-5 using a 'while' loop.

```
# program to print 1-5 using a 'while' loop.
i = 1
# while loop for i = 1 to 5
while i <= 5:
    print(i)
    i += 1
```

Output:

```
1
2
3
4
5
```

Explanation: The while loop continued as long as i is less than or equal to 5. The i += 1 statement increments the value of i by 1 on each iteration of the loop.

WHILE loop with ELSE:

- When the '**while**' condition becomes **False**, the loop checks for the **optional** '**else**' block.
  - If 'else' block is available, it executes the '**else**' block and then exits the loop.
  - If 'else' block is not available, then simply exits the loop.

Application: Write a program to print 1-5 using a 'while' loop with 'else'

```
# program to print 1-5 using a 'while' loop and 'else' block
i=1
# while loop for i = 1 to 5
while i <= 5:
    print(i)
    i += 1
else:
    print("Reached end of the loop")
```

Output:

```
1
2
3
```

```
4
5
Reached end of the loop
```

Explanation: The while loop continued until from 1 to 5. Once the loop is complete, the 'else' block is executed. Then, exited the loop.
The i += 1 statement increments the value of i by 1 on each iteration of the loop.

Application:

```
# Aim: Generate Fibonacci series up to a given number of terms
n = int(input("Enter how many Fibinacci terms : "))
i = 0

# Term1 and Term2
term1, term2 = 0, 1

# Is the nth term positive?
if n <= 0:
   print("Enter a positive integer>0.")

# If n is only 1 term
elif n == 1:
   print("Fibonacci series of",n,"terms is:")
   print(term1)

# Find and generate Fibonacci series up to n term
else:
   print("Fibonacci series of",n,"terms: ")
   while i < n:
       print(term1, end=" ")
       next = term1 + term2
       term1 = term2
       term2 = next
       i+=1
```

Output:

```
Enter how many Fibinacci terms: 5
Fibonacci series of 5 terms:
0 1 1 2 3
```

2. The ‘for’ loop:

Definition:

- A for loop is used to iterate over a sequence of elements such as string, range(), list, set, tuple or dictionary.
- The code inside the loop is executed repeatedly once for each element in the sequence.
- The “for” loop is used when we KNOW number of iterations.

Syntax: for

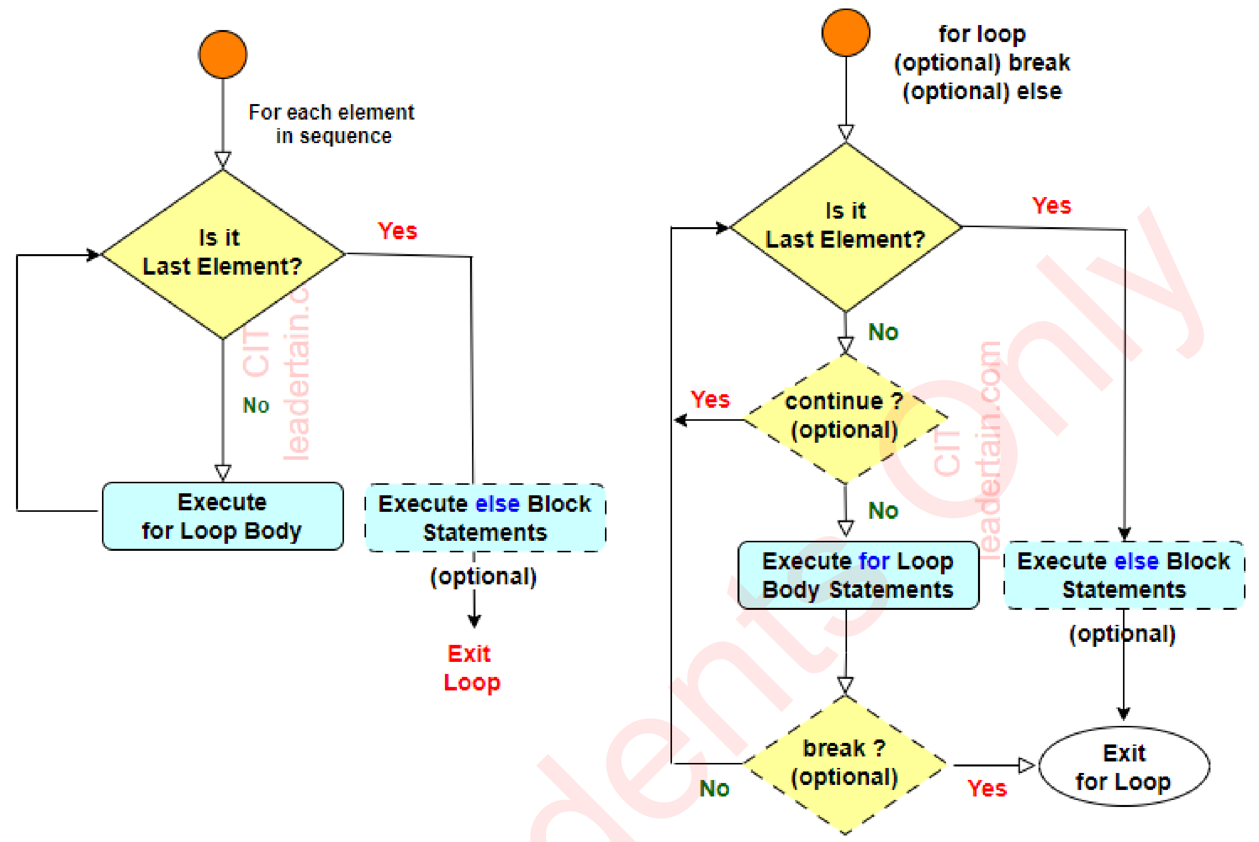
```
for var in sequence:
    Loop body statements

else:   (optional)
    Block of statements
```

```
for var in sequence:
  if (condition):
      continue (optional)
  if (condition):
      break    (optional)
  Loop body statements
else:   (optional)
  Block of statements
```

var - an iterator variable
sequence - a sequence of elements; a sequence can be a string, range(), list, set, tuple or dictionary
Loop body statements - a block of for loop statements
else - is an optional block in ‘**for**’. When the for loop completes, it enters **‘else’ block of statements**.

- **var** is an **iterator variable** that takes one element at a time from the **sequence** on each iteration.
- After taking the element in **var**, the **loop statements** execute.
- **for** loop continues until the last value of the sequence is reached.

Flow chart - for loop



Uses of for loop:

for loop through values or a string: iterates through each value or each character of a string sequence

```
# for loop using values sequences
print("Iterate data sequence")
for i in (10,20,30,40,50):  # for block is exected for each value
    print(i, end=' ')
else:
    print("\nEnd of the loop")
```

Output:

```
Iterate data sequence
10 20 30 40 50
End of the loop
```

```
# for loop using string sequence
print("Iterate string sequence")
for i in "CIT Python":      # for block is exected for each character
    print(i, end=' ')
```

Output:

```
Iterate string sequence
C I T   P y t h o n
```

for loop through a list: iterates through each element of a list/set/tuple/dict sequences

```
# for loop using list sequence
print("Iterate a list")
# for block is exected for each element of the list
branches=["CIT","CSE", "AI", "AIML","ECE"]
for i in branches:
    print(i, end=' ')
```

Output:

```
Iterate a list
CIT CSE AI AIML ECE
```

for loop using range() function: iterates through a range of values in sequence

```
# for loop using sequence of range()function
```

Method-1: range(end value)

Default start value is 0

Parameter - is the end value, (Goes up to end - 1, not including end value)

Default Increment by 1

```
print("Iterate in range(stop)")
for i in range(5):  # 0-4, the 5 not included
    print(i)
```

Output:

```
Iterate in range(stop)
0
1
2
3
4
```

Method-2: range(start, end value)

Parameter-1 is start value,

Parameter-2 is end value, (Goes up to end - 1, not including end value)

Default - Increment by 1

```
print("Iterate in range(start, stop)")    #
for i in range(1,5):  # 1-4, the 5 not included
    print(i)
```

Output:

```
Iterate in range(start, stop)
1
2
3
4
```

Method-3: **range(start, end, incr/decr value)** –

Parameter-1 is start value,

Parameter-2 is end value, (Goes up to end - 1, not including end value)

Parameter-3 is Increment or decrement value.

```
print("Iterate in range(start, stop, inc/dec)")
for i in range(1,5,2):  # 1,3 the 5 not included
    print(i)
```

Output:

```
Iterate in range(start, stop, inc/dec)
1
3
```

Application-1: Program to count number of even integers in the given list using for loop.

```
# for loop: Program to count the number of even integers in a list.
# List of integer numbers
numbers = [10, 5, 7, 4, 20, 37, 9]
# variable to track the even count
ecount = 0
# repete over the list
for n in numbers:
    if n % 2 == 0:
        ecount += 1
print("Count of even numbers is", ecount)
```

Output:

```
Count of even numbers is 3
```

Application-2: Write a program to Find GCD of 2 numbers

```
# Storing user input into num1 and num2
num1 = int(input("Enter integer number1 : "))
num2 = int(input("Enter integer number2 : "))
# identify smallest of 2 numbers and assign to limit
if(num1<num2):
    limit = num1
else:
    limit = num2
for i in range(1,limit+1):
    # Checks if the current value of i is
    # factor of both the integers num1 & num2
    if(num1%i==0 and num2%i==0):
        gcd = i
print(f"GCD of input numbers {num1} and {num2} is: {gcd}")
```

Output:

```
Enter integer number1 : 50
Enter integer number2 : 100
GCD of input numbers 50 and 100 is: 50
```

Application-3: Write a program to print ASCII value & character set in Python

```
print("ASCII ==> Character\n");
for i in range(0,127):
    print(f"{i} ==> {chr(i)}")
```

Output:

```
…
120 ==> x
121 ==> y
122 ==> z
123 ==> {
124 ==> |
125 ==> }
126 ==> ~
```

Calculating Running Total

Definition:
A running total is the sum of numbers that accumulates over a sequence of numbers.
To calculate a running total in Python, you can use a loop to iterate through a range or list of numbers and keep track of the running total using a variable.

For example, if you have a list of numbers [1, 2, 3, 4], the running total would be [1, 3, 6, 10], where each element is the sum of all the elements that came before it.

Purpose:
A running total provides subtotals for any further calculations or for preparing a report.

Application:

```
'''
Program to find running total within a given range (using for loop)
'''
runningTotal = 0;
num = int(input("Enter +ve integer 1-100 : "))
if(num<0 or num>100):
    print("Out of range")
    exit(0)

# for loop terminates when num is less than count
for i in range(num+1):
    runningTotal += i
    print(f"{i}   {runningTotal} ")
```

Output:

```
Enter +ve integer 1-100 : 5
0   0
1   1
2   3
3   6
4   10
5   15
```

Explanation:
The program takes an input number between 1 and 100 from the user. If the given number is outside the range of 1-100, then the programs exits. If the given number is with in the range of 1-100 then the running total is calculated in 'for' loop and prints the result for each iteration.

Input Validation Loop

Definition:
An input validation loop prompts the user to enter input data, checks the input for validity, and repeats the prompt until valid input is entered. The loop continues until the user enters valid input and then the program can proceed with the remaining steps.

Purpose:
Input validation loops in Python ensure that the user enters valid input data. This is important because invalid data can cause errors or unexpected behavior in the program.

Application:

```python
'''
Aim: Check the input number is valid. If invalid, then repeat the
prompt to reenter another number
'''
while True:
    user_input = input("Enter a number between 1 and 10 : ")
    num = int(user_input)
    if num < 1 or num > 10:
        print("Invalid number.")
    else:
        print("Valid number.")
        break
```

Output:

```
Enter a number between 1 and 10 : 27
Invalid number.
Enter a number between 1 and 10 : 7
Valid number.
```

Explanation:
In this example, the loop continues until the user enters a valid number between 1 and 10. The input is first converted to an integer using the int() function. If the input is an invalid integer, the loop continues. If the input is a valid and within the range, the loop is exited and the program can proceed with the remaining steps.

Nested Loops in Python

Definition:
Nested loop in Python is a loop that is placed inside another loop. The nested loops are used to iterate over multiple groups of data or to perform a task repeatedly for each element of multiple lists or collections of data.

We have **for** and **while** loops in Python. We can nest these loops in any combination in Python. Two such combinations are as follows:

- **for** loop nested with another **for loop,**
- **for** loop nested with a **while loop**

Purpose:
Nested loops are typically used for working with patterns, multidimensional data structures, such as printing two-dimensional arrays, iterating a list that contains a nested list.

General Syntax: Nested Loop in Python

```
OuterLoop Expression:

    InnerLoop Expression:
        Statements inside InnerLoop

    Statements inside Outer_Loop
```

Syntax: Nested for Loops

```
for outer_var in outer_sequence:

    for inner_var in inner_sequence:
        Statements in inner for loop

    Statements in outer for loop
```

Note: Each iteration of the outer for loop triggers a complete iteration of the inner for loop.

Syntax: Nested for - while Loops

```
for outer_var in outer_sequence:

    while (condition):
        Statements in inner while loop

    Statements in outer for loop
```

Note: Each iteration of the outer for loop triggers a complete iteration of the inner while loop.

Application-1:

```
# Lab-5: Use a for loop to print a triangle using *s.
# Allow the user to specify how high the triangle should be.
rows = int(input('Enter rows: '))
for i in range(1,rows+1):
    for j in range(0,i):
        print('*', end=' ')
    print('')
```

Output:

```
Enter rows: 5
*
* *
* * *
* * * *
* * * * *
```

Application-2:

```
#Program to print multiplication tables using nested "for" loop
# Outer loop - tables 5 and 6
for i in range(5, 7):
    # Inner loop from 1 to 10
    for j in range(1, 11):
        print(i, "*", j, "=", i*j)
    print()
```

Output:

```
5 * 1 = 5          6 * 1 = 6
5 * 2 = 10         6 * 2 = 12
5 * 3 = 15         6 * 3 = 18
5 * 4 = 20         6 * 4 = 24
5 * 5 = 25         6 * 5 = 30
5 * 6 = 30         6 * 6 = 36
5 * 7 = 35         6 * 7 = 42
5 * 8 = 40         6 * 8 = 48
5 * 9 = 45         6 * 9 = 54
5 * 10 = 50        6 * 10 = 60
```

Application-3:

```
#Program to print a number pattern using nested “while” loop
i=1
while i<=5:
    j=1
    while j<=i:
        print(j,end=" ")
        j=j+1
    print("")
    i=i+1
```

Output:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Application-4:

#Find Prime numbers in an Interval using nested “for - while” loop

```
lownum = int(input("Enter low number of interval : "))
highnum = int(input("Enter high number of interval : "))
print("Prime numbers between", lownum, "and", highnum, "are:")
for num in range(lownum, highnum + 1):
   # Primes are always > 1
   if num > 1:
        i = 2
        while (i<num):
            if (num % i) == 0:
                break
            i += 1
        else:
            print(num, end=’ ‘)
```

Output:

```
Enter low number of interval : 1
Enter high number of interval : 20
Prime numbers between 1 and 20 are:
2  3  5  7  11  13  17  19
```

Jump Statements in Python

The Jump Statements are loop Control Statements in Python. The loop Control Statements are used to change the normal flow of a loop based on a condition.

The 3 jump or loop control statements in Python are,

1. **break** statement
2. **continue** statement
3. **pass** statement

1. “break” statement

Definition:

- The break statement in Python is used to terminate the loop and brings the control out of the loop. The break statement is used in both the **while** and the **for** loops.
- The break statement is especially useful to quit from a nested loop (loop within a loop). It terminates the inner loop and control shifts to the statement in the outer loop.

Note: The “break” statement almost always needs an “if” condition to work properly.

Syntax:
break

Using break in while loop:

```
while (condition):
    #statements
    if (condition):
          break
    #statements
```

Using break in for loop:

```
for var in sequence:
    #statements
    if (condition):
          break
    #statements
```

Application:

BREAK in WHILE loop :

If the ‘**while**’ loop encounters an **optional** ‘**break**’, the loop simply **exits** even though the ‘**while**’ condition is **True**.

Application: Write a program to print 1-5 using a ‘while’ loop with ‘break’ to stop at 4.

```
i=1
# while loop with i = 1 to 3
while i <= 5:
    print(i)
    i += 1
    if(i==4):
        break
```

Output:

```
1
2
3
```

Explanation: The while loop continued until it encountered 4 and then exited while loop. The i += 1 statement increments the value of i by 1 on each iteration of the loop.

BREAK in FOR loop :

If the ‘**for**’ loop encounters an **optional** ‘**break**’, the loop simply **exits** even though the ‘for’ **sequence is not completed**.

Application: Write a program to print 1-5 using a ‘for’ loop with ‘break’ to stop at 4.

```
# for loop with i = 1 to 3
for i in range (1,6):
    if(i==4):
        break
    print(i)
```

Output:

```
1
2
3
```

Explanation: The ‘for’ loop continued until it encountered 4 and then exited ‘for’ loop.

2. **“continue” statement**

Definition:

The “continue” statement forces the control to **skip the current iteration** and go to the next iteration of the loop.

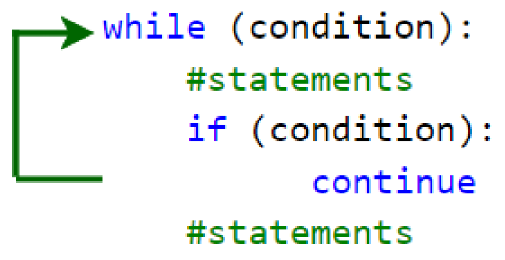
A. In “**while**” statement, the “**continue**” statement will directly **jump the execution control to “condition”**,

B. In “**for**” statement, the “**continue**” statement will **jump the execution control to the next element in the given sequence**.

Syntax:

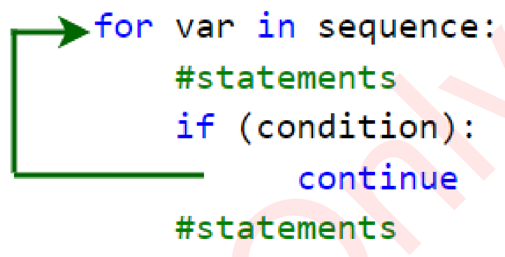
continue

Using continue in while loop:



```
while (condition):
        #statements
        if (condition):
                continue
        #statements
```

Using continue in for loop:



```
for var in sequence:
        #statements
        if (condition):
                continue
        #statements
```

Application:

CONTINUE in WHILE loop:
If the ‘**while**’ loop encounters an **optional** ‘**continue**’, the loop simply skips the current iteration and jumps to the ‘condition’ for next iteration.

Application: Program to print <u>even numbers between 1 and 5</u> using while loop & continue (skip) on odds

```
n = 1
while n < 5:
    n += 1
    if (n % 2) != 0:
        continue
    print(n)
```

Output:
```
2
4
```

Explanation: The while loop continued until 2. When it encountered 3, the value incremented to 4 and executed ‘continue’ to skip the iteration.

CONTINUE in FOR loop :
If the loop encounters an **optional** ‘**continue**’, the loop simply skips the current iteration and jumps to next iteration in the sequence.

Application: Program to print <u>even numbers between 1 and 5</u> using 'for' loop & continue (skip) on odds

```
for n in range(1,6):
    if (n % 2) != 0:
        continue
    print(n)
```

Output:
```
2
4
```

Explanation: The ‘for’ loop continued until 2. When it encountered 3, it executed ‘continue’ to skip 3 and continued with 4 in the sequence.

3. "pass" statement

Nothing happens when the "**pass**" statement is executed. Hence, it is a **null** operation and is considered a placeholder for future code.

- **Empty code is not allowed in loops, function definitions, class definitions, or if statements and causes errors in Python.**
- **So, "pass" statement is used to write empty loops, control statements, functions, or classes to avoid errors.**

Syntax:
`pass`

Application:

PASS in IF and WHILE loop
`n=1`
`while (n<5):`
`    if (n==3):`
`        pass`
`    n += 1`
	PASS in FOR loop
	`college = "Chalapathi"`
	`for i in college:`
	`    pass`

PASS in Function
`def func():`
`    pass`
`func()`
	PASS in Class
	`class name:`
	`    pass`

Quick Reference

Comparison of Loops

‘for’ loop	‘while’ loop
Pre-Test or Entry controlled loop - Checks for LAST ELEMENT at TOP	**Pre-Test or Entry** controlled loop - CONDITION is specified at TOP
Sequence controlled loop (Known number of elements)	**Event** (or Condition) controlled loop
Use it when you **know** how many times to iterate	Use it when you **don’t know** how many times to iterate
Repeats for all elements in a sequence, except the last one.	Repeats until a condition is met

Syntax:
for var in sequence:
    loop statements
else: (optional)
    block of statements
	Syntax:
	`Initialization (optional)`
	`while (Condition):`
	`  Loop Block of statements`
	`  Incr/Decr (optional)`
	`else:    (optional)`
	`  Block of statements`

`Example: for`
 	
`# 1-4, the 5 not included`
`for i in range(1,5):`
`    print(i)`
	`Example: while`
	`i = 1`
	`# while loop for i = 1 to 5`
	`while i <= 5:`
	`    print(i)`
	`    i += 1`

`Output:` 1 2 3 4	`Output:` 1 2 3 4 5

Comparison of “break” and “continue” statements

break	continue
Used to terminate the loop	Used to SKIP current iteration and go to NEXT iteration
Control passed to outside the loop	Control passed to the beginning of the loop for next iteration
EXIT from control loop	Loop takes NEXT iteration
“break” is used in LOOPS (for, while)	“continue” is used in LOOPS (for, while)

Syntax:
`for var in sequence:`
`#body of loop`
`  if(condition)`
`       break`
	Syntax:
	`for var in sequence:`
	`#body of loop`
	`  if(condition)`
	`       continue`

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

Output: 0 1 2	**Output:** 0 1 2 4 5

Comparison of "break", exit(), sys.exit(), quit()

break	exit ( ) or sys.exit()	quit ( )
break is a keyword in Python; therefore it can't be used as a variable name.	**exit()** is a standard library function in Python. exit can be used as a variable name. sys module can also be used: **import sys** **sys.exit()**	**quit()** is a standard library function in Python. quit can be used as a variable name.
break causes an immediate termination from a loop (for, while) and jumps to the remaining program.	exit() terminates whole program execution.	quit() terminates whole program execution.
break transfers the control to outside the loop (for, while).	exit() returns the control to the operating system or another program that uses this one as a sub-process.	

Example of break
// some code here before while loop
while(true)
  ...
  if(condition)
    break;
# some code after while loop 	
	Example of exit()
	// some code here before while loop
	while(true)
	  ...
	  if(condition)
	    exit()
	# some code after while loop
		Example of quit()
		// some code here before while loop
		while(true)
		  ...
		  if(condition)
		    quit()
		# some code after while loop

In the above code, break terminates the while loop and some code after the while loop will be executed after breaking the loop.	In the above code, when if(condition) returns True, exit() will be executed and the program will get terminated.	In the above code, when if(condition) returns True, quit() will be executed and the program will get terminated.
Conclusion: break is a statement that terminates loops and jumps to the next program statements.	**Conclusion:** exit() is a library function that causes the immediate termination of the entire program.	**Conclusion:** quit() is a library function that causes the immediate termination of the entire program.

Application: Scenario based solution using while-else loop

```
''' Scenario: A team of players play a game. There is a qualified
score per the game. Each players can score and add up to total scored.
Aim: Find whether the team scored more or less of the qualified score.
Use while loop.  '''
moreScores = 'yes'
totalScores = 0
player = 1
# Qualified Score per game
totalToQualify = int(input('What is the qualified score per game? '))
while moreScores == 'yes':
    # Get score per player
    scoresPerPlayer = int(input(f'Enter score for player {player}: '))
    totalScores += scoresPerPlayer
    player += 1
    # Ask if user wants to input another score
    moreScores = input('Do you want to enter score for another player? yes or
no : ')
else:
    print("*** End of the game ***")

#Calculate scores less/more than qualified score per game
if totalScores >= totalToQualify:
    print('Your team scored', abs(totalToQualify - totalScores), f'points
more than qualified score {totalToQualify} per game.')
elif totalScores <= totalToQualify:
    print(f'Your team scored', totalToQualify - totalScores, f'points less
than qualified score {totalToQualify} per game.')
```

Output:

```
What is the qualified score per game? 100
Enter score for player 1: 50
Do you want to enter score for another player? yes or no : yes
Enter score for player 2: 40
Do you want to enter score for another player? yes or no : yes
Enter score for player 3: 20
Do you want to enter score for another player? yes or no : no
*** End of the game ***
Your team scored 10 points more than qualified score 100 per game.
```

Section-3: Strings: Accessing characters and Substring in Strings, Data Encryption, Strings and Number Systems.

Accessing Characters and Substrings in Strings

Introduction:
A string in Python is an array of Unicode characters enclosed in quotes. Also, Python does not have a character data type; a single character is simply a string with a length of 1.
String indexing in Python is zero-based: the first character in the string has index 0 , the next has index 1, and so on.

Accessing Individual Characters:

In Python, we can access individual characters in a string using indexing. The characters in a string in Python can be accessed using both normal indexing and negative indexing.

➢ **Normal Indexing -** Each character in the string is assigned a numerical index starting from **0 to n-1**, where **n** is the length of the string. So characters in a string of size n, can be accessed from **0 to n-1**.

➢ **Negative Indexing -** A string will also have negative indexing. A negative index number starting from -1 is assigned from the last character in a string. So, **-1 for last character, -2 for 2nd from the last, -3 for 3rd from the last** and so on.

index	0	1	2	3	4	5	6	7
string	S	o	f	t	w	a	r	e
-ve Index	-8	-7	-6	-5	-4	-3	-2	-1

Individual characters in a string can be accessed by the **string name** followed by an **index number** in square brackets **[ ]**.

Syntax:

string_name [index]

Example:

```
# Accessing characters in strings
st = "Software Pros!"
print(st[0])   # Output: S
print(st[5])   # Output: a
print(st[0:4]) # Output: Soft
print(st[-1])  # Output: !
```

Explanation:
In the above example, the first line creates a string variable st. The next three lines demonstrate how to access individual characters in the string using indexing.

- 1st print statement outputs the character at index 0, which is 'S'.
- 2nd print statement outputs the character at index 5, which is 'a'.
- 3rd print statement outputs the characters between 0 and 3, 'Soft'. (not including index 4).
- 4th print statement uses a negative index value to access the last character in the string, which is '!'.

Accessing Substrings:

In Python, you can **access substrings** from a string by **using slicing**. Slicing allows us to extract a portion of the original string by using the starting and ending index values.

String slicing is the process of obtaining a range of characters or a substring of a string by using its indices. Following are the 2 methods to slice a string.

1. **Array slicing (: operator)**
2. **slice() function**

1. **Array slicing (: operator)**

Definition:
Array slicing is used to obtain a portion of a string array or a list. It uses the **slicing operator :** and square brackets to slice a string.

Syntax:

```
object [ start : stop : step ]
```

start - start index of the slice (included),
stop - end index of the slice (excluded), and
step - step size is the number of elements to skip between each element in the slice

Application on Array Slicing	**Application Find Palindrome**

`s = "COLLEGE"`
`print(s[1:6])   # OLLEG    6 excluded`
`print(s[1:6:2]) # OLG      6 excluded`
`print(s[:3])    # COL      3 excluded`
`print(s[5:])    # GE       5 to last`
`# Negative index`
`print(s[-4:-1])     # LEG  -1 excluded`
`print(s[1:-4])      # OL   -4 excluded`
`print(s[5:1:-2]) #GL, in Reverse order`
`# Reverse`
`print(s[::-1])  # EGELLOC  String Reverse`
	`st1 = input("Enter a string : ")`
	`st2 = st1[:: - 1]`
	`if(st1 == st2):`
	`   print("This string is a Palindrome")`
	`else:`
	`   print("This string is not a Palindrome")`
	
	`Ex: level, madam, mom`

Table shows how the string sequence is sliced using **: operator**

Index	0	1	2	3	4	5	6
s	C	O	L	L	E	G	E
s[1:6]	C	O	L	L	E	G	E
s[1:6:2]	C	O	L	L	E	G	E
s[:3] s[0:end]	C	O	L	L	E	G	E
s[5:] s[beg :]	C	O	L	L	E	G	E

+ve index	0	1	2	3	4	5	6
-ve Index	-7	-6	-5	-4	-3	-2	-1
`s[-4:-1]`	C	O	L	L	E	G	E
`s[1:-4]`	C	O	L	L	E	G	E
`s[5:1:-2]`	C	O	L	L	E	G	E

`Reverse a string`							
`s[::-1]`	E	G	E	L	L	O	C

2. slice() Function

Definition:

1. The **slice() returns a slice object** (a portion size)
2. The **slice object is used as an index to slice** a sequence such as string, list, tuple, or range.

Syntax:

slice (start , stop , step)

start - start index of the slice (included),
stop - end index of the slice (excluded), and
step - step size is the number of elements to skip between each element in the slice

Application: slice():

```
# slice() function
s = "Our CIT College!"
sub = slice(0, 3)    # Creates a slice object representing [0:3]
result = s[sub]      # Slices the string s using the slice object sub
print(result)       # Output: "Our"
```

Output: Our

String format methods

String formatting is the process of inserting a custom string or variable in predefined text. Python allows string formatting using one of the following 5 methods.

1. **% (String Format Operator)**
2. **format() method**
3. **f-strings**
4. **Built-in methods**
5. **String Template Class** (external module: **from string import Template**)

1. % (String Format Operator):

The % Operator is called a String Format Operator or an Interpolation Operator. It is used for simple positional formatting in strings. It allows you to insert values into a string, replacing placeholders with actual values. The placeholders are represented by percent signs followed by a format specifier that defines the type of the value being inserted.

Syntax:

```
<"format specifiers">  %  <data/vars>
```

- **format specifiers** - carries any string with %formatSpecifiers as placeholders (%d, %f, %s)
- **' % '** is the **String Format Operator** that substitutes data/variable value into format specifier
- **data/vars** - values to replace format specifiers

<"format specifiers"> may have format specifiers with Padding for data values as specified below:

%<fieldwidth>.<precision>f	**%6.2f**
%<fieldwidth>d	**%3d**
%<fieldwidth>s	**%10s**

<fieldwidth> is the total number of digits given for the value
<precision> is the number of decimal digits out of the given total digits
The unfilled digit positions will be added as padding spaces on the left.

Example:

```
name = "Raj"
age = 25
marks = 75.55
# without padding
print("Name:%s, Age:%d, Marks:%f" % (name, age, marks))
```

Output: Name:Raj, Age:25, Marks:75.550000

```
# with padding
print("Name:%10s, Age:%3d, Marks:%6.2f" % (name, age, marks))
```

Output:
```
Name:       Raj, Age: 25, Marks: 75.55
```

Table: List of format specifiers in Python

Format Specifier	Conversion
%c	character
%s	string conversion via str() prior to formatting
%i	signed decimal integer
%d	signed decimal integer
%u	unsigned decimal integer
%o	octal integer
%x	hexadecimal integer (lowercase letters)
%X	hexadecimal integer (UPPERcase letters)
%e	exponential notation (with lowercase 'e')
%E	exponential notation (with UPPERcase 'E')
%f	floating point real number
%g	the shorter of %f and %e
%G	the shorter of %f and %E

2. format() method

The **format()** method formats the given values and insert them at placeholders in a string. The placeholders are represented by curly braces **{ }**.

Syntax1: `using format() with sequence of vars/values`

```
.format(var0,var1...)
print("{},{} ".format(var0,var1))
```

- format() method must be preceded by **.** operator
- var1, var2,...var-n are variables or values we pass into format() method
- placeholder { } is a value specifier.
    - Each pair of {}s represents a value from the variable passed into format()
    - The sequence of variables in the format() method must match the sequence of { } in quotes

Example:

```
name = "Venkat"
age = 20
print("My name is {} and I am {} years old.".format(name,age))
```

Output:
My name is Venkat and I am 20 years old.

Syntax2: `using format() with index number of vars/values`

```
.format(var0,var1...)
print("{var index0},{var index0},{var index1}".format(var0,var1))
```

- format() method must be preceded by **.** operator
- var1, var2,...var-n are variables or values we pass into format() method
- Each variable is indexed starting from 0 and increments by 1
- `{var index#}` represents the value of the variable specified in that position in the format(var0, var1, var2, ...) function.
- The index/position of variables in format() function starts with 0 and increments by 1

Example:

```
name = "Venkat"
age = 20
grade = 'A'
print("{0} has grade {2}. {0} is {1} years old.".format(name,age,grade))
```

Output:
Varsha has grade A. Venkat is 20 years old.

3. **3. f string format**

   a. **f** or **F** means formatted strings that are more readable and faster. (>= 3.6).
   b. To create an f-string, prefix the string with letter “**f**”.
   c. These f strings contain replacement fields in curly braces **{ }**
   d. The f or F in front of strings tells Python to look at the values, expressions, or objects inside { } and substitute them with the values of the given variables or expressions.
   e. Formatted strings are evaluated at run time (while other string literals always have a constant value).

Example1: Basic fstrings

```
name1 = "Divya"
name2 = "Nitin"
cash1=5000
cash2=7000
total_cash = cash1 + cash2
#print in format method-2: Better one
print(f"Cash from {name1} = {cash1}")
print(f"Cash from {name2} = {cash2}")
print(f"Total amount = {total_cash}")
```

Output:
```
Cash from Nitin = 100
Cash from Naveen = 200
Total amount = 300
```

Example2: f string for precision, datetime and number conversion

```
import decimal
import datetime

# precision: nested fields, output: 12.35
width = 12
precision = 4
value = decimal.Decimal("12.3456789")
print(f"result:{value:{width}.{precision}}" )
print(f"result:{value:{2}.{5}}" )
# date format specifier, output: March 27, 2017
today = datetime.datetime(year=2023, month=3, day=17)
print(f"{today:%B %d, %Y}" )
```

```
# hex integer format specifier, output: 0x400
number = 1024
print(f"{number:#0x}" )
```

These are commonly used string format approaches in Python. We can customize the string format using different arguments and formatting options.

4. Built-in methods to format strings

In Python, the **class 'str'** provides several built-in methods to format or convert strings. The following table shows these methods and how they format the strings when they are used with a string object.

Table: Built-in methods to format strings

Method	Description	s=”software Engineers”
s.capitalize()	converts the first character to uppercase.	Software Engineers
s.upper()	Converts all the characters in a string to uppercase.	SOFTWARE ENGINEERS
s.lower()	Converts all the characters in a string to lowercase.	software engineers
s.isupper()	Returns True if all the characters are uppercase. Otherwise, False	False
s.islower()	Returns True if all the characters are lowercase. Or else False.	False
s.find(substring, [start, end])	Returns the index of a specified character in the string or the start position of the given substring.	s.find(“Eng”) 9
s.count(substring,[start,end])	Counts the occurrence of a character or substring in a string.	s.count(“r”) 2
s.expandtabs([tabsize])	Replaces tabs defined by \t with spaces. Default tabsize = 8	
s.endswith(substring,[start, end])	Returns True if a string ends with the specified substring. False otherwise.	s.endswith(“neers”) True

s.startswith(substring, [start, end])	Returns True if a string starts with the specified substring. False otherwise.	s.startswith("Soft") True
s.isalnum()	Return True if all characters in a string are alphanumeric. False otherwise.	False
s.isalpha()	Return True if all characters in a string are alphabetic. False otherwise.	True
s.isdigit()	Return True if all characters in a string are digits. False otherwise.	False
s.split([separator],[maxsplit])	Splits a string separated by a separator(defaults is whitespace) and an optional **maxsplit** to determine the split limit. Returns a list.	["Software","Engineers"]

sep.join(sequence)	Takes all items in an iterable sequence (list, tuple, string), separates them by a given separator, and Joins them into a single string.	sep="_"
		seq="CIT"
		sep.join(seq) => C_I_T

s.replace(old, new,[maxreplace])	Replace old substring contained in the string s with a new substring.	s.("Engineers","Programmer") Software Programmers
s.swapcase()	Returns a new string with swapped case. i.e., uppercase becomes lowercase and vice versa.	sOFTWARE pROGRAMMERS
s.strip([characters])	Removes whitespaces or optional characters at the beginning and at the end of the string.	
s.lstrip([characters])	Removes leading whitespace or optional characters from a string.	
s.rstrip([characters])	Removes trailing spaces at the end of the string.	

Application: Using Built-in format methods

```
# built-in methods to format strings in class 'str'
s = "Software Pros"
print("capitalize:",s.capitalize() )
print("upper:",s.upper() )
print("lower:",s.lower() )
print("isupper:",s.isupper() )
print("islower:",s.islower() )
print("index# find:",s.find("Pros") )
print("count:",s.count("r") )
print("isnum:",s.isalnum() )
print("isalpha:",s.isalpha() )
print("isdigit:",s.isdigit() )
print("split:",s.split() )
print("join:",  "-".join(s) )
print("replace:",s.replace("Pros","Engineers"))
print("swapcase:",s.swapcase() )
```

Output:

```
capitalize: Software pros
upper: SOFTWARE PROS
lower: software pros
isupper: False
islower: False
index# find: 9
count: 2
isnum: False
isalpha: False
isdigit: False
split: ['Software', 'Pros']
join: S-o-f-t-w-a-r-e- -P-r-o-s
replace: Software Engineers
swapcase: sOFTWARE pROS
```

Operators for String Operations

Python provides the following operators for string operations:

- **String concatenation operator “ + ”**
- **String repetition operator “ * ”**
- String Slicing operator “ : ” to obtain substrings (See String slicing, p44)
- Indexing to traverse through strings (See Accessing Individual Character, p43)
- Membership operators (in, not in) to search for strings (See Operators in Unit-I)
- Relational operators (>, >=, <, <=) to compare strings (See Comparing Strings, p13)

Here, we will discuss + and * operators.
The + operator is used to concatenate 2 or more strings into one string.
The * operator is used to repeat a string up to a given number of times.

Operator	Purpose	Operation	Description
+	Concatenation	s1 + s2	Concatenates two strings, s1 and s2.
*	Repetition	s * n	Makes n copies of string s.

(+) Concatenation Operator:

Definition:

The + operator is used to join or concatenate two strings.
This concatenation operator in Python concatenates only objects of the same type.

Usage:

```
concatenate_string = string1 + string2   # concatenate the two strings
```

(*) Repetition Operator:

Definition:

The * operator is used to repeat a given string n number of times (similar to multiplication).

Usage:

```
repeat_string = string1 * n  # repeats string1 n times
```

Application:

```
# Concatenate & Repetition of strings
s1 = "Computer "
s2 = "Science"
s3 = s1 + s2
print(s3)

s4 = s1*3
print(s4)
```

Output:

```
Computer Science
Computer Computer Computer
```

String padding functions in Python

Definition:

In Python, String padding functions add extra characters such as spaces or zeros, at the start or end of a string to get a required length. Python does provide several built-in string padding functions for this purpose.

The commonly used string padding functions are,

1. **ljust(),**
2. **rjust(), and**
3. **center().**

Purpose:

These methods are very useful for formatting text in the form of tables or displaying information in a fixed-width format.

1. **ljust()**

Syntax:

```
svar.ljust(width[, fillchar])
```

ljust() function returns left-justified string of given width. The string is padded with **fillchar** (default is space) to make up the length.

Example:

```
s = 'Guntur'
padded_s = s.ljust(10, '*')
print(padded_s)    # Guntur****
```

2. **rjust()**

Syntax:

```
svar.rjust(width[, fillchar])
```

rjust() function returns right-justified string of given width. The string is padded with **fillchar** (default is space) to make up the length.

Example:

```
s = 'Guntur'
padded_s = s.rjust(10, '*')
print(padded_s)    # ****Guntur
```

3. **center()**

Syntax:

```
svar.center(width[, fillchar])
```

center() function returns centered string in the given width. The string is padded with **fillchar** (default is space) to make up the length.

Example:

```
s = 'Guntur'
padded_s = s.center(10, '*')
print(padded_s)    # **Guntur**
```

Data Encryption

Definition:
The process of converting information that cannot be understood by the unauthorized user is called data encryption. The reverse process is called decryption. Data encryption is used to protect the information transmitted over the network. The encrypted data prevents data corruption, sniffing, stealing, or security attacks.

The network protocols such as FTPS and HTTPS do provide security to the information transmitted over the network.

Security attacks:
Any action or a breach that compromises the security of information owned by an individual or an organization is called a security attack. Security attacks are classified into two: **Passive** and **Active**

➢ **Passive Attacks** - Unauthorized persons secretly reading or listening to private messages or message patterns while transmitting between a sender and a receiver.
➢ **Active Attacks** - Modification of the original data stream or the creation of a false data stream. Also includes,
  - Masquerade - one entity pretends to be a different entity
  - Replay- Passively capture and Unauthorized retransmission
  - **DOS (Denial Of Service)** - Disruption of an entire network

Process of Data Encryption:
- The information that is to be transmitted is called '**Plain Text**'.
- The **sender encrypts** the message by translating it into a secret code called '**Cipher Text**'.
- The **receiver decrypts** the cipher text into the original message or plain text.
- Both parties use **secret keys (public key & private key)** to encrypt and decrypt messages.
- **Caesar cipher** is a simple encryption method that has been in use for thousands of years.

Caesar cipher Encryption:

- Letter in a given plain text is changed to a letter that appears a certain number of positions farther down the alphabet set.
- For the characters near the end, the method goes back to the beginning of the alphabet set to locate the replacement characters.
- For example, if the distance value of a Caesar cipher is right-shift by 2 characters, the string "day" would be encrypted as "fca"

a	b	c	**d**	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	**y**	z
c	d	e	**f**	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	**a**	b

Application: Caesar cipher encryption

```
# Caesar Cypher Encryption - Method1

msg = input('Enter your message: ')

dist= int(input('Enter cipher distance: '))

cmsg=""

for ch in msg:

    ordnum=ord(ch)

    ciphernum=ordnum+dist

    if ciphernum>ord('z'):

        ciphernum=ord('a')+dist-(ord('z')-ordnum+1)

    cmsg=cmsg+chr(ciphernum)

print(cmsg)
```

```
# Caesar Cypher Encryption - Method2
msg = input('Enter your message: ')
dist= int(input('Enter cipher distance: '))
cmsg=""
for ch in msg:
    # Add space for space
    if ch==" ":
        cmsg+=" "
    # uppercase encryption
    elif (ch.isupper()):
       cmsg+=chr((ord(ch)+dist-65)%26+65)
    # lowercase encryption
    else:
        cmsg+=chr((ord(ch)+dist-97)%26+ 97)
print(cmsg)
```

Output:

```
Enter your message: day
Enter cipher distance: 2
fca
```

Application: Caesar cipher decryption

```
# Caesar Cypher Decryption

code=input('Enter your text: ')

dist=int(input('Enter distance: '))

msg=""

for ch in code:

    ordnum=ord(ch)

    ciphernum=ordnum-dist

    if ciphernum<ord('a'):

        ciphernum=ord('z')-(dist-(ord('a')-ordnum+1))

    msg=msg+chr(ciphernum)

print(msg)
```

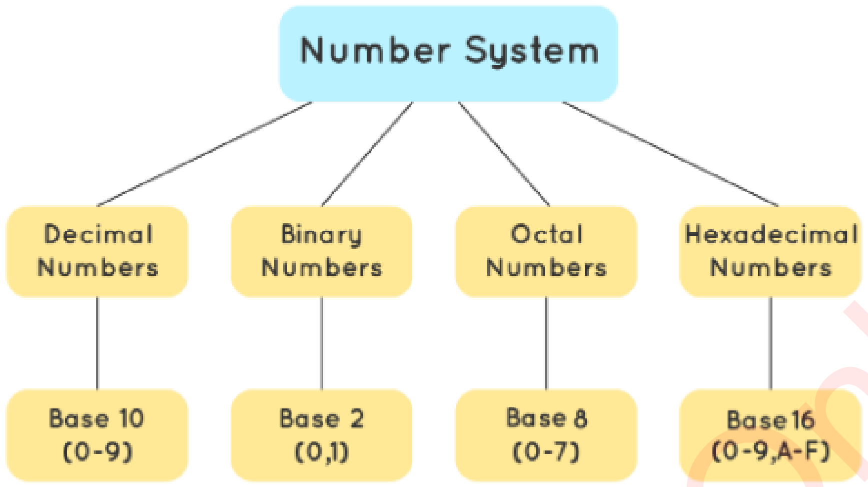
Output:

```
Enter your text: fca
Enter distance: 2
day
```

Number Systems

Number systems are the technique to represent numbers in the computer system architecture, every value that we save or read has a defined number system.
Computer architecture supports the following number systems.

1. **Binary number system**
2. **Octal number system**
3. **Decimal number system**
4. **Hexadecimal (hex) number system**



1) Binary Number System (Base: 2, Digits: 0, 1)
A Binary number system has only two digits 0 and 1. All binary numbers are represented in 0s and 1s.

2) Octal number system (Base: 8, Digits: 0-7)
Octal number system has only 8 digits from 0 to 7. All octal numbers are represented in 0,1,2,3,4,5,6 and 7.

3) Decimal number system (Base: 10, Digits: 0-9)
Decimal number system has only 10 digits from 0 to 9. All decimal numbers are represented in 0,1,2,3,4,5,6, 7,8, and 9.

4) Hexadecimal number system (Base: 16, Digits: 0-9, A-F)
A Hexadecimal number system has 16 alphanumeric values from 0 to 9 and A to F. All hexadecimal numbers are represented in 0,1,2,3,4,5,6, 7,8,9,A,B,C,D,E, and F. Here A is 10, B is 11, C is 12, D is 13, E is 14 and F is 15.

Table: Number Systems & Representation in Python

Number system	Base	Digits used	Example	Python assignment
Binary	**2**	**0,1**	**$(11110000)_2$**	**var = 0b11110000**
Octal	**8**	**0,1,2,3,4,5,6,7**	**$(360)_8$**	**var = 0o360**
Decimal	**10**	**0,1,2,3,4,5,6,7,8,9**	**$(240)_{10}$**	**var = 240**
Hexadecimal	**16**	**0,1,2,3,4,5,6,7,8,9, A,B,C,D,E,F**	**$(F0)_{16}$**	**var = 0xF0**

Decimal to Binary Conversion:

- **Manual conversion** - Decimal number is divided by 2 until we get 1 or 0 as the final remainder.
  $28_{10} = 11100_2$

Base target	Decimal	Remainder
2	28	**0**
2	14	**0**
2	7	**1**
2	3	**1**
2	**1**	

Decimal to Octal Conversion:

- **Manual conversion** - Decimal number is divided by 8 until we get **0 to 7** as the final remainder.
  $28_{10} = 34_8$

Base target	Decimal	Remainder
8	28	**4**
8	**3**	

Decimal to Hexadecimal Conversion:

- **Manual conversion** - Decimal number is divided by 16 until we get **0 to 15** as the final remainder.
  $28_{10} = 1C_{16}$

Base target	Decimal	Remainder
16	28	**12 = C**
16	**1**	

Automatic conversion: Decimal to Binary, Octal, Hexadecimal
From decimal to binary, octal or hexadecimal, use **bin( ), oct(), hex()** functions respectively.
From binary, octal or hexadecimal to decimal, use **int(other num, base)** function..

Application:

```
# Aim: Program to convert Decimal to Binary, Octal and Hexadecimal
# Decimal to Binary, Octal, Hexadecimal
n = 28
bn = bin(n)
ot = oct(n)
hx = hex(n)
print("Decimal to Binary ", n, "=", bn)
print("Decimal to Octal ", n, "=", ot)
print("Decimal to Hexadecimal ", n, "=", hx)

#Binary to Decimal
print("Binary to Decimal = ",int(bn,2))
#Octal to Decimal
print("Octal to Decimal = ",int(ot,8))
#Hexadecimal to Decimal
print("Hexa to Decimal = ",int(hx,16))
```

Output:

```
Decimal to Binary  28 = 0b11100
Decimal to Octal  28 = 0o34
Decimal to Hexadecimal  28 = 0x1c
Binary to Decimal =  28
Octal to Decimal =  28
Hexa to Decimal =  28
```

Binary to Decimal Conversion

Binary Number = **11100** $_2$

1	**1**	**1**	**0**	**0**
$1x2^4$	**$1x2^3$**	**$1x2^2$**	**$0x2^1$**	**$0x2^0$**
16	**8**	**4**	**0**	**0**

= 16 + 8 + 4 + 0 + 0

Decimal number = 28 $_{10}$

Octal to Decimal Conversion

Octal Number is : **34** $_8$

3	4
$3x8^1$	$4x8^0$
24	4

= 24 + 4

Decimal number = 28 $_{10}$

Hexadecimal to Decimal Conversion

Hexadecimal Number is : **1c** $_{16}$

1	c = 12
$1x16^1$	$12x16^0$
16	12

= 16 + 124 + 4

Decimal number = 28 $_{10}$