



Unit-02.1

Python Operators



Prof. Jayesh D. Vagadiya

Computer Engineering Department

Darshan Institute of Engineering & Technology, Rajkot

✉ Jayesh.vagadiya@darshan.ac.in

☎ 9537133260





Outline

- ✓ Arithmetic operators
- ✓ Assignment operators
- ✓ Comparison operators
- ✓ Logical operators
- ✓ Identity operators
- ✓ Membership operators
- ✓ Bitwise operators

Operators in Python

- Operators are used to performing **operations** on variables and values.
- We can **segregate** python operators in following groups,
 - Arithmetic operators
 - Assignment operators
 - Comparison operators
 - Logical operators
 - Bitwise operators
 - Identity operators
 - Membership operators

Arithmetic Operators

- Arithmetic operators can be used with **numeric values** or variables to perform common **mathematical** operations
- Note : consider A = 10 and B = 3

Operator	Description	Example	Output
+	Addition	A + B	13
-	Subtraction	A - B	7
/	Division	A / B	3.3333333333333335
*	Multiplication	A * B	30
%	Modulus return the remainder	A % B	1
//	Floor division returns the quotient	A // B	3
**	Exponentiation	A ** B	10 * 10 * 10 = 1000

- Imp. Point: Python does not support pre/post increment(++)/decrement(--) operators

Assignment Operators

- Assignment Operators are used to **assigning values to variables**.
- Note : consider $A = 3$, $B = 5$ and $C = 0$

Operator	Description	Example	Output
=	Assign	$C = A + B$	8
+=	Add and Assign	$A += B$	8
-=	Subtract and Assign	$A -= B$	-2
*=	Multiply and Assign	$A *= B$	15
/=	Divide and Assign	$A /= B$	0.6
%=	Modulus and Assign	$A \% = B$	3
//=	Divide(floor) and Assign	$A // = B$	0

Relational Operators

- Relational operators are used for **comparing the values**. It either returns **True** or **False** according to the condition.
- Note : consider $A = 9, B = 5$

Operator	Description	Example	Output
>	Greater than	$A > B$	True
<	Less than	$A < B$	False
==	Equal to	$A == B$	False
!=	Not Equal to	$A != B$	True
>=	Greater than or equal to	$A >= B$	True
<=	Less than or equal to	$A <= B$	False

Logical Operators

- Logical Operators are used to perform **logical operations** on the values of variables. The value is either **true** or **false**. We can figure out the conditions by the result of the **truth values**.
- Note : consider A = 10 and B = 3

Operator	Description	Example	Output
and	Returns True if both statements are true	A > 5 and B < 5	True
or	Returns True if one of the statements is true	A > 5 or B > 5	True
not	Negate the result, returns True if the result is False	not (A > 5)	False

Bitwise Operators

- The bitwise operators are used to performing **bitwise calculations** on integers. The integers are first converted **into binary** and then operations are performed on **bit by bit**, hence the name bitwise operators. Then the result is returned in **decimal format**.
- Note : consider A = 10(binary - 1010) and B = 4(binary - 0100)

Operator	Description	Example	Output
A	Bitwise AND	A & B	0
	Bitwise OR	A B	14
~	Bitwise NOT	~A	-11
^	Bitwise XOR	A ^ B	14
>>	Bitwise right shift	A>>	5
<<	Bitwise left shift	A<<	20

Example

Example.py

```
1 x , y = 5,2
2 a,b = True,False
3
4 #Arithmetic operator
5 print('x // y =',x//y)
6 print('x ** y =',x**y)
7
9 #Assignemnt Operator
10 x+=5 # x = 10
11 y+=2 # y = 4
12 print('x y =',x,y)
13
14 #Comparison Operator
15 print('x >= y is',x>=y)
16 print('x <= y is',x<=y)
17
18 #Logical Operator
19 print('x and y is',a and b)
20 print('x or y is',a or b)
21 print('not x is',not a)
22
23 #Bitwise Operators
24 print('x & y is',x & y)
25 print('x | y is',x | y)
```

Output

```
x // y = 2
x ** y = 25
x y = 10 4
x >= y is True
x <= y is False
x and y is False
x or y is True
not x is False
x & y is 0
x | y is 14
```

Identity & Member Operators

□ Identity Operator

□ Note : consider A = [1,2], B = [1,2] and C=A

Operator	Description	Example	Output
is	Returns True if both variables are the same object	A is B A is C	False True
is not	Returns True if both variables are different object	A is not B	True

□ Member Operator

□ Note : consider A = 2 and B = [1,2,3]

Operator	Description	Example	Output
in	Returns True if a sequence with the specified value is present in the object	A in B	True
not in	Returns True if a sequence with the specified value is not present in the object	A not in B	False

Example

Example.py

```
1 x2 = 'Hello'
2 y2 = 'Hello'
3 x3 = [1,2,3]
4 y3 = x3
5
6 x = 'Hello world'
7 y = {1:'a',2:'b'}
9 z = [1,2,3,4,5]
10
11 # Identity Operator
12 print("x2 is y2 = ",x2 is y2)
13 print("x3 is y3 = ",x3 is y3)
14 print("x2 is not y2 =",x2 is not y2)
15
16 # Membership Operator
17 print("'H' in x = ", 'H' in x)
18 print("'hello' not in x = ", 'hello' not in x)
19 print("5 in z = ", 5 in z)
20 print("1 in y = ", 1 in y)
```

Output

```
x2 is y2 = True
x3 is y3 = True
x2 is not y2 = False
'H' in x = True
'hello' not in x = True
5 in z = True
1 in y = True
```



Unit-02.2

Conditional and Looping Statements



Prof. Jayesh D. Vagadiya

Computer Engineering Department

Darshan Institute of Engineering & Technology, Rajkot

✉ Jayesh.vagadiya@darshan.ac.in

☎ 9537133260





Outline

- ✓ If statement
- ✓ if-else statement
- ✓ nested if statement
- ✓ elif statement
- ✓ For loop statement
- ✓ While loop statement
- ✓ break
- ✓ continue
- ✓ pass keywords

Introduction

- The statements that help us to control the flow of execution in a program called **control statements**.
- There are two types of control statements.
- **Branching Statement :**
 - The statements that help us to select some statements for execution and skip other statements.
 - These statements are also called as **decision making statements** because they make decision based on some condition and select some statements for execution.
 - If statement
 - if-else statement
 - nested if statement
 - elif statement
- **Looping statements:**
 - The statements that help us to execute set of statements repeatedly are called as looping statements.
 - For loop statement
 - While loop statement

If statement

- if statement is written using the **if** keyword followed by **condition** and **colon(:)** .
- Code to execute when the condition is true will be ideally written in the next line with **Indentation** (white space).
- Python relies on indentation to define scope in the code (Other programming languages often use curly-brackets for this purpose).

Syntax

```
1 if some_condition :  
2     # Code to execute when condition is true
```

if statement ends with :

Indentation (tab/whitespace) at the beginning

ifdemo.py

```
1 x = 10  
2  
3 if x > 5 :  
4     print("X is greater than 5")
```

Output

```
1 X is greater than 5
```

If else statement

- This is basically a “two-way” decision statement. This is used when we must choose between two alternatives.

Syntax

```
1 if some_condition :  
2     # Code to execute when condition is true  
3 else :  
4     # Code to execute when condition is false
```

ifelsedemo.py

```
1 x = 3  
2  
3 if x > 5 :  
4     print("X is greater than 5")  
5 else :  
6     print("X is less than 5")
```

Output

```
1 X is less than 5
```


Example of if else

Example1.py

```
1 # Program checks if the number is positive or negative
2 num = int(input("Enter Number="))
3 if num >= 0:
4     print("Positive number")
5 else:
6     print("Negative number")
```

Output

```
Enter Number = 5
Positive number
```

Example2.py

```
1 # Program checks if the number is Odd or Even
2 num = int(input("Enter Number="))
3 if num % 2 == 0:
4     print("Even number")
5 else:
6     print("Odd number")
```

Output

```
Enter Number = 4
Positive number
```

Nested if statement

□ An if-else statement is written within **another if-else** statement is called nested if statement.

Syntax

```
1 if some_condition1 :  
2     if some_condition2 :  
3         # Code to execute when condition is true  
4     else :  
5         # Code to execute when condition is false
```

Output

```
Enter a Number = 4  
Positive number
```

Example1.py

```
1 num = float(input("Enter a number: "))  
2 if num >= 0:  
3     if num == 0:  
4         print("Zero")  
5     else:  
6         print("Positive number")  
7 else:  
8     print("Negative number")
```

Example

Example1.py

```
1 # find maximum number from three number
2 a = int(input("Enter A="))
3 b = int(input("Enter B="))
4 c = int(input("Enter C="))
5
6 if a>b:
7     if a>c:
8         g=a
9     else:
10        g=c
11 else:
12     if b>c:
13         g=b
14     else:
15         g=c
16
17 print("Greater = ",g)
```

Output

```
Enter A=10
Enter B=9
Enter C=2
Greater = 10
```

If, elif and else statement

- ❑ The **elif** is short for else if. It allows us to check for multiple expressions.
- ❑ If the condition for if is **False**, it checks the condition of the **next elif** block and so on.
- ❑ If all the conditions **are False**, the body of **else** is executed.
- ❑ Only **one block** among the several if...elif...else blocks is executed according to the condition.
- ❑ The if a block can have only one else block, but it can have **multiple elif** blocks.
- ❑ This statement is alternative for nested if statement to **overcome** the **complexity problem** involved in nested if statement.

If, elif and else statement

Syntax

```
1 if some_condition_1 :  
2     # Code to execute when condition 1 is true  
3 elif some_condition_2 :  
4     # Code to execute when condition 2 is true  
5 else :  
6     # Code to execute when both conditions are false
```

ifelifdemo.py

```
1 x = 10  
2  
3 if x > 12 :  
4     print("X is greater than 12")  
5 elif x > 5 :  
6     print("X is greater than 5")  
7 else :  
8     print("X is less than 5")
```

Output

```
1 X is greater than 5
```

Example

Example1.py

```
1 # Python Program to find Student Grade
2 english = float(input(" Please enter English Marks: "))
3 math = float(input(" Please enter Math score: "))
4 computers = float(input(" Please enter Computer Marks: "))
5 physics = float(input(" Please enter Physics Marks: "))
6 chemistry = float(input(" Please enter Chemistry Marks: "))
7
8 total = english + math + computers + physics + chemistry
9 percentage = (total / 500) * 100
10
11 print("Total Marks = %.2f" %total)
12 print("Marks Percentage = %.2f" %percentage)
13
14 if(percentage >= 90):
15     print("A Grade")
16 elif(percentage >= 80):
17     print("B Grade")
18 elif(percentage >= 70):
19     print("C Grade")
20 elif(percentage >= 60):
21     print("D Grade")
22 elif(percentage >= 40):
23     print("E Grade")
24 else:
25     print("Fail")
```

Output

```
Please enter English Marks: 50
Please enter Math score: 50
Please enter Computer Marks: 50
Please enter Physics Marks: 50
Please enter Chemistry Marks: 50
Total Marks = 250.00
Marks Percentage = 50.00
E Grade
```

For loop in python

- Many objects in python are **iterable**, meaning we can iterate **over every element** in the object.
 - such as every elements from the List, every characters from the string etc..
- We can use for loop to execute block of code for each element of **iterable** object.

Syntax

```
1 for temp_item in iterable_object :  
2     # Code to execute for each object in iterable
```

For loop ends with :

Indentation (tab/whitespace) at the beginning

fordemo1.py

```
1 my_list = [1, 2, 3, 4]  
2 for list_item in my_list :  
3     print(list_item)
```

Output :

1
2
3
4

fordemo2.py

```
1 my_list = [1,2,3,4,5,6,7,8,9]  
2 for list_item in my_list :  
3     if list_item % 2 == 0 :  
4         print(list_item)
```

Output :

2
4
6
8

range() function

- The range() function returns a **sequence of numbers**, starting from **0 by default**, and increments **by 1 (by default)**, and stops **before a specified number**.

Syntax

```
1 range(start, stop, step)
```

Example1.py

```
1 x = range(3, 6)
2 for n in x:
3     print(n)
```

Output

```
3
4
5
```

Example2.py

```
1 x = range(3, 20, 2)
2 for n in x:
3     print(n)
```

Output

```
3
5
-
19
```

Example3.py

```
1 x = range(20)
2 for n in x:
3     print(n)
```

Output

```
0
1
-
19
```

Example4.py

```
1 x = reversed(range(20))
2 for n in x:
3     print(n)
```

Output

```
19
18
-
0
```


For loop (tuple unpacking)

- Sometimes we have nested data structure like **List of tuples**, and if we want to iterate with such list we can use **tuple unpacking**.

withouttupleunpacking.py

```
1 my_list = [(1,2,3), (4,5,6), (7,8,9)]
2 for list_item in my_list :
3     print(list_item[1])
```

Output :

2
5
8

withtupleunpacking.py

```
1 my_list = [(1,2,3), (4,5,6), (7,8,9)]
2 for a,b,c in my_list :
3     print(b)
```

This
technique is
known as
tuple
unpacking

Output :

2
5
8

For loop with else

- The else block just after for/while is executed only when the loop is **NOT terminated** by a break statement.

Example1.py

```
1 for i in range(1, 4):
2     print(i)
3 else:
4     print("No Break")
```

Output

```
1
2
3
No Break
```

Example2.py

```
1 for i in range(1, 4):
2     print(i)
3     break
4 else: # Not executed as there is a break
5     print("No Break")
```

Output

```
1
```

Example

Example1.py

```
1 #odd numbers between 1 to n
2 num = int(input("Enter Number = "))
3 for i in range(1,num+1):
4     if i % 2 != 0:
5         print(i)
```

Output

```
Enter Number = 5
1
3
5
```

Example2.py

```
1 #series 1 + 4 + 9 + 16 + 25 + 36 + ...n
2 num = int(input("Enter Number = "))
3 sum = 0
4 for i in range(1,num+1):
5     sum = sum + i**2
6 print("Sum=",sum)
```

Output

```
Enter Number = 10
Sum= 385
```

Example

Example3.py

```
1 # Find factorial of the given number
2 num = int(input("Enter Number = "))
3 fact = 1
4 for i in range(1,num+1):
5     fact = fact * i
6 print("Factorial =", fact )
```

Output

```
Enter Number = 5
Factorial = 120
```

Example4.py

```
1 #series 1 - 2 + 3 - 4 + 5 - 6 + 7 ... n
2 num = int(input("Enter Number = "))
3 sum = 0
4 for i in range(1,num+1):
5     if i % 2 == 0:
6         sum = sum - i
7     else:
8         sum = sum + i
9 print("Sum=",sum)
```

Output

```
Enter Number = 10
Sum= -5
```

While loop

□ While loop will continue to **execute block of code** until some condition **remains True**.

□ For example,

- while felling hungry, keep eating
- while have internet pack available, keep watching videos

Syntax

```
1 while some_condition :  
2     # Code to execute in loop
```

while loop ends with :

Indentation (tab/whitespace) at the beginning

whiledemo.py

```
1 x = 0  
2 while x < 3 :  
3     print(x)  
4     x += 1    # x++ is invalid in python
```

Output :

0
1
2

withelse.py

```
1 x = 5  
2 while x < 3 :  
3     print(x)  
4     x += 1    # x++ is invalid in python  
5 else :  
6     print("X is greater than 3")
```

Output :
X is greater
than 3

Example

Example1.py

```
1 #odd numbers between 1 to n
2 num = int(input("Enter Number = "))
3 i=0
4 while(i<=num):
5     if i % 2 != 0:
6         print(i)
7     i+=1
```

Output

```
Enter Number = 5
1
3
5
```

Example2.py

```
1 #series 1 + 4 + 9 + 16 + 25 + 36 + ...n
2 num = int(input("Enter Number = "))
3 sum = i = 0
4 while(i<=num):
5     sum = sum + i**2
6     i+=1
7 print("Sum=",sum)
```

Output

```
Enter Number = 10
Sum= 385
```

Exercise programs

- Do Following programs using for and while loop.
 - WAP to find out sum of first and last digit of a given number
 - WAP to find whether the given number is prime or not.
 - WAP to find out prime numbers between given two numbers
 - WAP to print given number in reverse order.
 - WAP to check whether the given number is Armstrong or not.
 - WAP to find the sum of $1 + (1+2) + (1+2+3) + (1+2+3+4) + \dots + (1+2+3+4+\dots+n)$.

break keyword

- ❑ **Breaks** out of the current **closest enclosing loop**.
- ❑ Break Statement is a loop control statement that is used to **terminate the loop**.
- ❑ As soon as the break statement is encountered from within a loop, the loop iterations stop there, and control returns from the loop immediately to the **first statement** after the loop.
- ❑ Basically, break statements are used in situations when we are not sure about the actual number of iterations for the loop or we want to **terminate the loop based on some condition**.

Syntax

```
1 break
```

breakdemo.py

```
1 for temp in range(5) :  
2     if temp == 2 :  
3         break  
4  
5     print(temp)
```

Output :

```
0  
1
```


continue keyword

- ❑ Goes to the top of the **current closest enclosing loop**.
- ❑ Continue statement is a loop control statement that forces to execute the **next iteration** of the loop while **skipping the rest** of the code inside the loop for the current iteration only i.e.
- ❑ When the continue statement is executed in the loop, the code inside the loop following the continue statement will be **skipped** for the **current iteration** and the next iteration of the loop will begin.

Syntax

```
1 continue
```

continuedemo.py

```
1 for temp in range(5) :  
2     if temp == 2 :  
3         continue  
4  
5     print(temp)
```

Output :

```
0  
1  
3  
4
```

pass keyword

- ❑ Does nothing at all, will be used as a **placeholder** in conditions where you don't want to write anything.
- ❑ The pass statement is a **null statement**. But the difference between pass and comment is that comment is ignored by the interpreter whereas pass is not ignored.

Syntax

```
1 pass
```

passdemo.py

```
1 for temp in range(5) :  
2     pass
```

Output : (nothing)



Unit-02.3

Functions



Prof. Jayesh D. Vagadiya

Computer Engineering Department

Darshan Institute of Engineering & Technology, Rajkot

✉ Jayesh.vagadiya@darshan.ac.in

☎ 9537133260





Outline

- ✓ Creating function
- ✓ DOCSTRING
- ✓ Types of arguments
- ✓ Calling function
- ✓ return statement
- ✓ Scope of Variables
- ✓ Lambda expression
- ✓ Recursion

Functions in python

- ❑ Creating **clean repeatable code** is a key part of becoming an effective programmer.
- ❑ A function is a block of code which only **runs when it is called**.
- ❑ In Python a function is defined using the **def** keyword:

Syntax

```
def function_name() :  
    #code to execute when function is called
```

ends with :

Indentation (tab/whitespace) at the beginning

Functiondemo.py

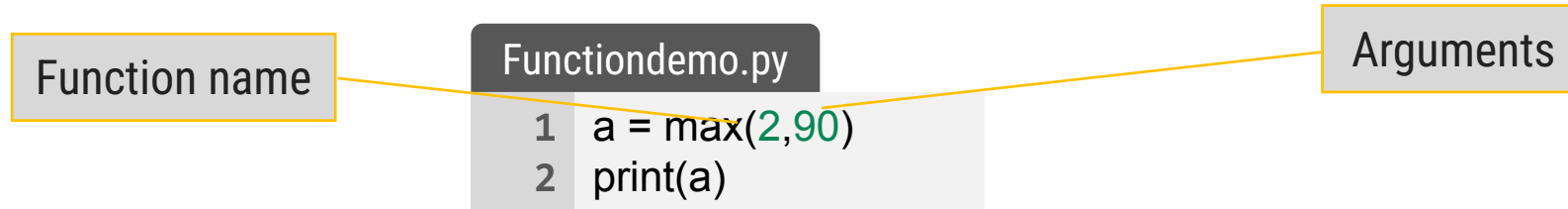
```
1 def seperator() :  
2     print('=====')  
3  
4 print("hello world")  
5 seperator()  
6 print("from darshan college")  
7 seperator()  
8 print("rajkot")
```

Output :
hello world

```
=====  
from darshan college  
=====  
rajkot
```

Function (cont.)

- There are two kinds of functions in Python.
- **Built-in functions** that are provided as part of Python - input(), print()...
- Functions that we define ourselves and then use
- We treat the of the built-in function names as **"new" reserved** words (i.e. we avoid them as variable names)
- In Python a function is some **reusable code** that takes **arguments(s)** as input does some computation and **then returns a result(s)**
- We call/invoke the function by using the function name, parenthesis and arguments in an expression



Function Names

- ❑ Use a **naming scheme**, and use it **consistently** (camel case syntax).
- ❑ For all names, avoid **abbreviations**, unless they are both standardized and widely used.
- ❑ The name should describe the **data's meaning** rather than its **type** (e.g., amount_due rather than money),
- ❑ Functions and methods should have names that **say what they do** or what they return (depending on their emphasis), but never how they do it—since that might change.
- ❑ All three functions below return the index position of the first occurrence of a name in a list of names, starting from the given starting index and using an algorithm that assumes the list is already sorted.

Functiondemo.py

```
1 def find(l, s, i=0): # BAD
2 def linear_search(l, s, i=0): # BAD
3 def first_index_of(sorted_name_list, name, start=0): # GOOD
```

Function (cont.) (DOCString & return)

- Doc string helps us to define the documentation about the function within the function itself.

Syntax

```
def function_name():  
    '''  
        DOCSTRING: explains the function  
        INPUT: explains input  
        OUTPUT: explains output  
    '''  
    #code to execute when function is called
```

Enclosed within triple quotes

- Unlike conventional source code comments, the docstring should describe what the function does, not how.
- The docstrings can be accessed using the `__doc__` method of the object or using the help function.

Help.py

```
1 Function_name. __doc__
```


Function Types

- ❑ Four kinds of functions can be created in Python: **global functions, local functions, lambda functions, and methods.**
- ❑ **Global function**
 - ❑ Global objects (including functions) are **accessible to any code in the same module** (i.e., the same .py file) in which the object is created.
- ❑ **Local functions**
 - ❑ (also called **nested functions**) are functions that are **defined inside** other functions. These functions are visible only to the function where they are defined;
 - ❑ They are especially useful for **creating small helper** functions that have no use elsewhere.
- ❑ **Lambda functions**
 - ❑ Lambda functions are expressions, so they can be created at their point of use;
 - ❑ however, they are much more limited than normal functions.
- ❑ **Methods**
 - ❑ *Methods* are functions that are associated with a particular **data type** and can be used only in conjunction with the data type (when we cover object-oriented programming.)

Exercise

- ❑ WAP to count simple interest using function.
- ❑ WAP to find maximum number from given two numbers using function.
- ❑ WAP that defines a function exchange to interchange the values of two variables, say x and y.

Function Arguments

□ A function by using the following types of formal arguments:

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

□ Required arguments

- Required arguments are the arguments passed to a function in **correct positional** order.
- During a function call, values passed through arguments should be in the **order of parameters** in the function definition.

Demo.py

```
1 def add_number(n1,n2):  
2     print("Sum = ", n1+n2)  
3  
4 add_number(2,3)  
5 add_number(3,5)
```

Output

```
Sum = 5  
Sum = 8
```

Function Arguments(cont.)

□ Keyword arguments

- When you use keyword arguments in a function call, the caller identifies the arguments by the **parameter name**.
- This allows you to **skip arguments or place them out of order** because the Python interpreter is able to use the keywords provided to match the values with parameters.
- Functions can also be called using keyword arguments of the form **kward=value**
- During a function call, values passed through arguments need not be in the **order of parameters** in the function definition. This can be achieved by keyword arguments.

Demo.py

```
1 def subtract_number(n1,n2):  
2     print("Subtraction = ", n1-n2)  
3  
4 subtract_number(20,10)  
5 subtract_number(n2=20,n1=10) Keyword Arguments
```

Output

```
Subtraction = 10  
Subtraction = -10
```

Function Arguments(cont.)

□ Default arguments

- A default argument is an argument that **assumes a default value** if a value is not provided in the function call for that argument.
- Default arguments are values that are provided while **defining functions**.
- The assignment **operator =** is used to assign a default value to the argument.
- Default arguments become **optional during the function calls**.
- If we provide a value to the default arguments during function calls, it overrides the default value.
- Default arguments should follow **non-default** arguments.

Demo.py

```
1 def add_number(n1,n2 = 10):  
2     print("Sum = ", n1+n2)  
3  
4 add_number(2,3)  
5 add_number(3)
```

Output

```
Sum = 5  
Sum = 13
```

Function Arguments(cont.)

□ Variable-length arguments

- Variable-length arguments are also known as **arbitrary arguments**. If we don't know the number of arguments needed for the function in advance, we can use **arbitrary arguments**.
- There are **two types** of variable length arguments

□ Arbitrary positional arguments

- You may need to process a function for more arguments than you specified while **defining the function**.
- These arguments are called *variable-length* arguments and are not **named in the function definition**, unlike required and default arguments.

Syntax

```
def functionname([formal_args,] *var_args_tuple ):  
    #code to execute when function is called
```

Tuple representing
variable length arguments

- An **asterisk (*)** is placed before the variable name that will hold the values of all non keyword variable arguments.
- This tuple remains empty if no additional arguments are specified during the function call

Function Arguments(cont.)

□ Arbitrary positional arguments example

Demo.py

```
1 def add_number(n1,*n):
2     sum = n1
3     for i in n:
4         sum = sum + i
5     print("Sum = ", sum)
6
7 add_number(2)
8 add_number(10,10,10,10)
```

Output

```
Sum = 2
Sum = 40
```

Function Arguments(cont.)

□ Arbitrary keyword arguments

- For arbitrary positional argument, a double asterisk (**) is placed before a parameter in a function which can hold keyword variable-length arguments.

Demo.py

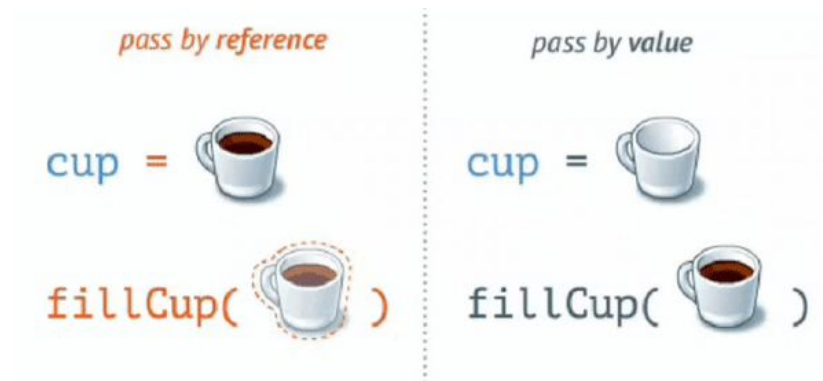
```
1 def add_Number(**a):
2     sum = 0
3     sum = a['a'] + a['b']
4     print("Sum=",sum)
5     for i in a.items():
6         print(i)
7     print(a)
8
9 add_Number(a=4,b=3,c=4)
```

Output

```
Sum= 7
('a', 4)
('b', 3)
('c', 4)
{'a': 4, 'b': 3, 'c': 4}
```


Pass by reference vs value

- ❑ In call by value method, the value of the **actual parameters is copied** into the formal parameters.
- ❑ In call by reference, the **address of the variable** is passed into the function call as the **actual parameter**.
- ❑ In python value is passed as “**Call by Object Reference**” or “**Call by assignment**”.
- ❑ When we pass whole numbers, strings or tuples to a function, the passing is like **call-by-value** because you **can not change the value of the immutable** objects being passed to the function.
- ❑ Whereas passing mutable objects can be considered as call by reference because when their values are **changed inside the function**, then it will also be **reflected outside the function**.



Pass by reference vs value

CallByValue.py

```
1 # Call by Value
2 def modify_String(s):
3     s = "Darshan University"
4     print("Inside Function:", s)
5
6
7 str = "Darshan"
8 modify_String(str)
9 print("Outside Function:", str)
```

Output

```
Inside Function: Darshan University
Outside Function: Darshan
```

CallByReference.py

```
1 # Call by Reference
2 def add_more(list):
3     list.append(50)
4     print("Inside Function:", list)
5
6
7 list = [1,2,3,4,5]
8 add_more(list)
9 print("Outside Function:", list)
```

Output

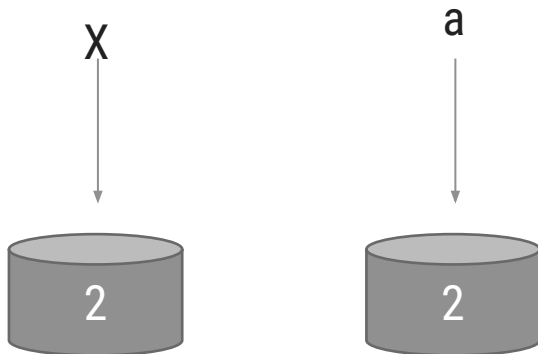
```
Inside Function: [1, 2, 3, 4, 5, 50]
Outside Function: [1, 2, 3, 4, 5, 50]
```

Pass by reference vs value

```
def printA(a):  
    a = 3
```

```
x=2  
printA(x)  
print("A = ",x)
```

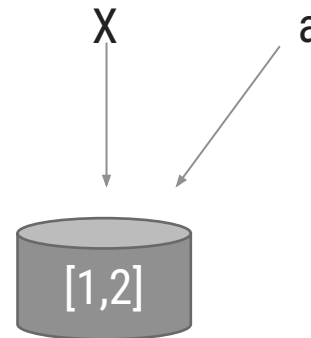
Call by Value



```
def printList(a):  
    a.append(3)
```

```
x= [1,2]  
printList(x)  
print("List",x)
```

Call by Reference



return Statement

- ❑ **return statement** : return allows us to assign the output of the **function to a new variable**, return is use to send back the **result of the function**, instead of just **printing it out**.
- ❑ A return statement is used to **end the execution** of the function call. The statements after the return statements are **not executed**.
- ❑ If the return statement is without any expression, then the special value **None is returned**.
- ❑ In python we can return multiple values from function using **object, tuple, list, Dictionary**.

returnDemo.py

```
1 def add_number(n1,n2) :  
2     return n1 + n2  
3  
4 sum1 = add_number(5,3)  
5 sum2 = add_number(6,1)  
6 print(sum1)  
7 print(sum2)
```

Output :

8
7

return Multiple values

returndemo.py

```
1 def test():
2     a=1
3     b=2
4     c=3
5     return a,b,c # return multiple values
6
7 x = test()
8 a,b,c = test()
9 print(x)
10 print(x[1])
11 print("a, b, c=",a,b,c)
```

Output

```
(1, 2, 3)
2
a, b, c= 1 2 3
```

Exercise

- ❑ WAP that define a function to find factorial of given number.
- ❑ WAP that defines a function which returns 1 if the number is prime otherwise return 0.
- ❑ WAP to generate Fibonacci series of N given number using function name fibbo.
- ❑ WAP that defines a function to add first n numbers.

Scope of Variables

- All variables in a program **may not be accessible at all locations** in that program. This depends on where you have **declared a variable**.
- The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python:
 - Global variables
 - Local variables
- Variables that are defined inside a **function body have a local scope**, and those defined outside have a **global scope**.
- This means that local variables can be accessed **only inside the function** in which they are declared.
- whereas global variables can be accessed throughout the **program body by all functions**.
- When we call a function, the variables declared inside it are **brought into scope**.

Scope of Variables (cont.)

Local.py

```
1 total = 0
2 def sum( arg1, arg2 ):
3     total = arg1 + arg2
4     print ("Inside the function local total : ", total)
5     return total;
6 # Now you can call sum function
7 sum( 10, 20 );
8 print ("Outside the function global total : ", total)
```

Output

```
Inside the function local total :  30
Outside the function global total :  0
```

Global.py

```
1 total = 0
2 def sum( arg1, arg2 ):
3     global total
4     total = arg1 + arg2
5     print ("Inside the function local total : ", total)
6     return total;
7 # Now you can call sum function
8 sum( 10, 20 );
9 print ("Outside the function global total : ", total)
```

Output

```
Inside the function local total :  30
Outside the function global total :  30
```


Lambda Function

- ❑ We can use the **lambda keyword** to create small anonymous functions.
- ❑ These functions are called anonymous because they are not declared in the standard manner by using the **def keyword**.
- ❑ Lambda forms can take any number of arguments but return **just one value in the form of an expression**. They cannot contain **multiple expressions**.
- ❑ An anonymous function cannot be a direct call to **print** because lambda requires an expression.
- ❑ Lambda functions have their own **local namespace and cannot access variables** other than those in their parameter list and those in the global namespace.
- ❑ Although it appears that lambda's are a one-line version of a function, they are not equivalent to **inline statements** in C or C++

Lambda Function (cont.)

Syntax

```
1 lambda arg1,arg2..argN: expression
```

- The parameters are **optional**, and if supplied they are normally just comma- separated variable names, that is, **positional arguments**.
- Lambda functions accept all kinds of arguments, just like normal def function
- The expression can **not contain branches or loops** (although conditional expressions are allowed), and cannot have a return statement. The result of a lambda expression is an **anonymous function**.
- When a lambda function is called it **returns the result of computing the expression** as its result.

Example.py

```
1 sum = lambda arg1,arg2: arg1 + arg2
2 print("Total",sum(5,5))
```

Output

Total 10

Lambda Function (cont.)

Cube.py

```
1 def cube(y):
2     return y*y*y
3 lambda_cube = lambda y: y*y*y
4
5 # using the normally defined function
6 print(cube(5))
7 # using the lambda function
8 print(lambda_cube(5))
```

Output

```
125
125
```

WithIfElse.py

```
1 # Example of lambda function using if-else
2 Max = lambda a, b : a if(a > b) else b
3 print(Max(1, 2))
```

Output

```
2
```

call.py

```
1 # Lambda functions can be Immediately Invoked
2 print((lambda x: x*x)(3))
```

Output

```
9
```

Exercise

- ❑ WAP to find square of given number using lambda expression.
- ❑ WAP to find simple interest using lambda function.
- ❑ WAP to make simple calculator using lambda expression.

Map

- `map()` function returns a map **object(which is an iterator)** of the results after applying the given function to each item of a given **iterable (list, tuple etc.)**

Syntax

```
1 map(fun, iterable)
```

demo1.py

```
1 def addition(n):  
2     return n * n  
3  
4 numbers = [1, 2, 3, 4]  
5 result = list(map(addition, numbers))  
6 # result = list(map(lambda n:n*n,numbers))  
7 print(result)
```

Output

```
[1, 4, 9, 16]
```

demo2.py

```
1 strdata = ["Darshan", "University"]  
2 lengthdata = list(map(len, strdata))  
3 print(lengthdata)
```

Output

```
[7, 10]
```

Functional Programming Tools: filter and reduce

- ❑ To find out items based on a test function we can use a Filter.
- ❑ For example, the following filter call picks out items in a sequence that are divide by 2:

demo1.py

```
1 lista = [1,2,3,4,5,6,7,8,9,10]
2 ans = list(filter(lambda x: x % 2 == 0 ,lista))
3 print(ans)
```

Output

```
[2, 4, 6, 8, 10]
```

Functional Programming Tools: filter and reduce

- Apply functions to pairs of items and running results (reduce(fun,seq)).
 - In the first step, the first two elements of the sequence are chosen, and the result is obtained.
 - The result is then stored after applying the same function to the previously obtained result and the number just succeeding the second element.
 - This process is repeated until there are no more elements in the container.
 - The final result is returned and printed to the console.

demo1.py

```
1 from functools import reduce
2 lista = [1,2,3,4,5,6,7,8,9,10]
3 ans = reduce(lambda a,b : a+b,lista)
4 print(ans)
```

Output

55

demo2.py

```
1 from functools import reduce
2 lista = [1,2,3,4,5,6,7,8,9,10]
3 ans = reduce(lambda a,b : a if a>b else b,lista)
4 Print(ans)
```

Output

10

Recursion

- ❑ Any function which calls **itself** is called recursive function and such function calls are called recursive calls.
- ❑ Recursion cannot be applied to **all problems**, but it is more useful for the tasks that can be defined in terms of a **similar subtask**.
- ❑ It is idea of representing **problem a with smaller problems**.
- ❑ Any problem that can be solved recursively can be solved **iteratively**.
- ❑ When recursive function call itself, the memory for called function allocated and different copy of the local variable **is created for each function call**.
- ❑ Some of the problem best suitable for recursion are
 - ❑ Factorial
 - ❑ Fibonacci
 - ❑ Tower of Hanoi

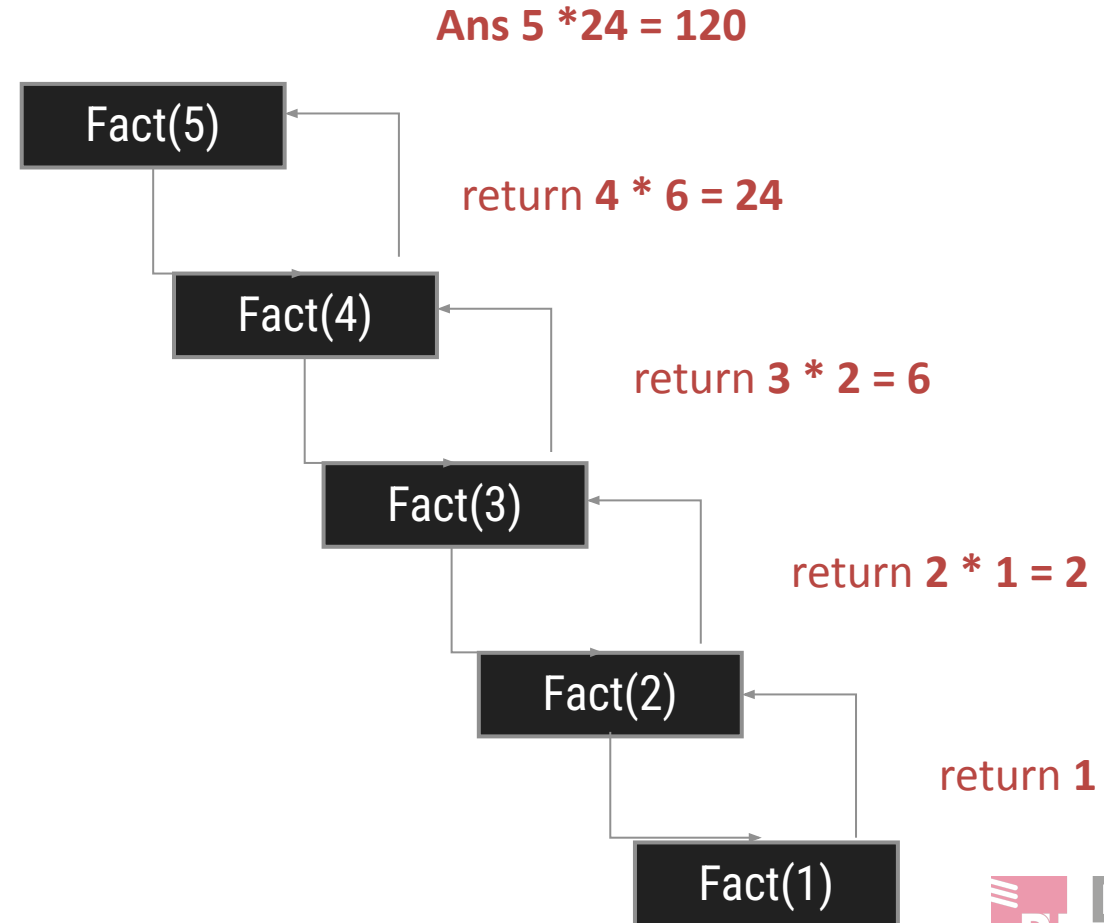
Properties of Recursion

- ❑ A recursive function can **go infinite like a loop**. To avoid infinite running of recursive function, there are two properties that a recursive function must have.
- ❑ **Base Case or Base criteria**
 - ❑ It allows the recursion algorithm to stop.
 - ❑ A base case is typically a problem that is small enough to solve directly.
- ❑ **Progressive approach**
 - ❑ A recursive algorithm must change its state in such a way that it moves forward to the base case.

Recursion - factorial example

- The factorial of a integer n , is product of
 - $n * (n-1) * (n-2) * \dots * 1$
- Recursive definition of factorial
 - $n! = n * (n-1)!$
 - Example
 - $3! = 3 * 2 * 1$
 - $3! = 3 * (2 * 1)$
 - $3! = 3 * (2!)$

Recursive trace



Examples

Factorial.py

```
1 def recursive_factorial(n):
2     if n == 1:
3         return n
4     else:
5         return n * recursive_factorial(n-1)
6     # recursive function call
7
8 # user input
9 num = int(input("Enter the number="))
10 print("Factorial of number", num, "=", recursive_factorial(num))
```

Output

Enter the number=5
Factorial of number 5 = 120

fibonacci.py

```
1 def r_fibonacci(n):
2     if n <= 1:
3         return n
4     else:
5         return(r_fibonacci(n-1) + r_fibonacci(n-2))
6 n = 8
7 print("Fibonacci series:")
8 for i in range(n):
9     print(r_fibonacci(i))
```

Output

Fibonacci series:
0
1
1
-
-
13

Exercise

- Write a program to find factorial of a given number using recursion.
- WAP to convert decimal number into binary using recursion.
- WAP to use recursive calls to evaluate $F(x) = x - x^3/3! + x^5/5! - x^7/7! + \dots + x^n/n!$