Python

What is Python ?

Python is a clear and powerful object-oriented programming language, comparable to Perl, Ruby, or Java.

Python is a programming language that combines features of C and Java.

https://www.python.org/

History

- Python was developed by *Guido Van Rossum* in the year *1990* at Stichting Mathematisch Centrum in the Netherlands as a successor of a language called ABC.
- Name “Python” picked from TV Show Monty Python’s Flying Circus.

https://docs.python.org/3/license.html

Version

- Python 0.9.0 - February, 1991
- Python 1.0 - January 1994
- Python 2.0 - October, 2000
- Python 3.0 - December, 2008
- Python 3.1 - June, 2009
- Python 3.2 - February, 2011
- Python 3.3 - September, 2012
- Python 3.4 - March, 2014
- Python 3.5 - September, 2015
- Python 3.6 - December, 2016
- Python 3.7 - June, 2018

Python 3.5+ cannot be used on Windows XP or earlier.

https://www.python.org/doc/versions/

Features

- Easy to Learn
- High Level Language
- Interpreted Language
- Platform Independent
- Procedure and Object Oriented
- Huge Library
- Scalable

Application for Python

- Web Application - Django, Pyramid, Flask, Bottle
- Desktop GUI Application – Tkinter
- Console Based Application
- Games and 3D Application
- Mobile Application
- Scientific and Numeric
- Data Science
- Machine Learning - scikit-learn and TensorFlow
- Data Analysis - Matplotlib, Seaborn
- Business Application

https://www.python.org/about/apps/

Byte Code – Byte Code represents the fixed set of instruction created by Python developers representing all type of operations like arithmetic operations, comparison operation, memory related operation etc.

The size of each byte code instruction is 1 byte or 8 bits.

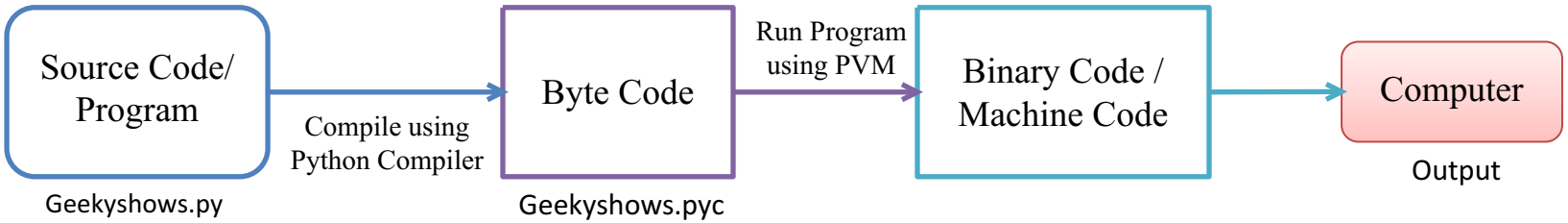
We can find byte code instruction in the .pyc file.

Python Compiler – A Python Compiler converts the program source code into byte code.

Type of Python Compilers :-

- CPython
- Jpython/ Jython
- PyPy
- RubyPython
- IronPython
- StacklessPython
- Pythonxy
- AnacondaPython

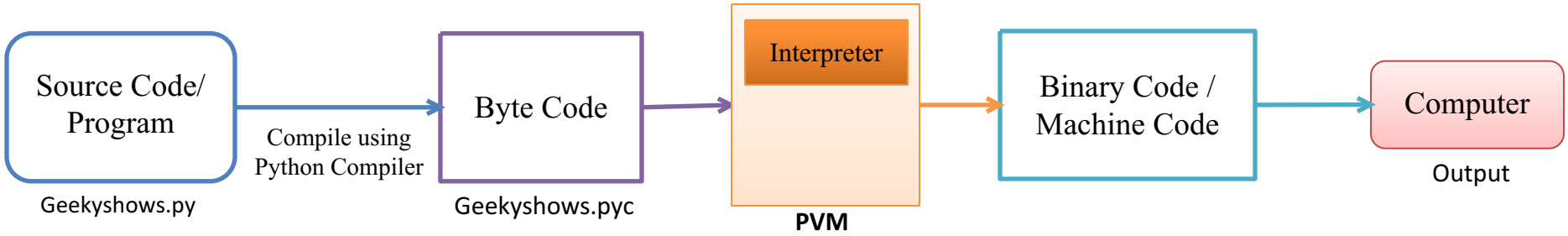
- Write Source Code / Program
- Compile the Program using Python Compiler
- Compiler Converts the Python Program into byte Code
- Computer/Machine Can not understand Byte Code so we convert it into Machine Code using PVM
- PVM uses an interpreter which understands the byte code and convert it into machine code
- Machine Code instructions are then executed by the processor and results are displayed



Python Virtual Machine

Python Virtual Machine (PVM) is a program which provides programming environment. The role of PVM is to convert the byte code instructions into machine code so the computer can execute those machine code instructions and display the output.

Interpreter converts the byte code into machine code and sends that machine code to the computer processor for execution.



Executing Python Program

- Command Line Window
- IDLE
- Notepad or Notepad++
- PyCharm
- Visual Studio Code

Identifier

An identifier is a name having a few letters, numbers and special characters _ (underscore).
It should always start with a non-numeric character.
It is used to identify a variable, function, symbolic constant, class etc.
Ex : -

X2
PI
Sigma
matadd
full_name

- Python is case sensitive programming language.

  d is not equal to D

  t is not equal to T

  rahul is not equal to Rahul

  rahul is not eqaul to RAHUL

Keywords or Reserved Words

Python language uses the following keywords which are not available to users to use them as identifiers.

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Constants

A constant is an identifier whose value cannot be changed throughout the execution of a program whereas the variable value keeps on changing.

There are no constants in Python, the way they exist in C and Java.

In Python, It is not possible to define constant whose value can not be changed.

In Python, Constants are usually defined on a module level and written in all capital letters with underscores separating words but remember its value can be changed.

Ex:-

PI

TOTAL

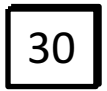
MIN_VALUE

Variable

In C, Java or some other programming languages, a variable is an identifier or a name, connected to memory location.

a = 30

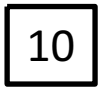
a



12114

b = 10

b



12115

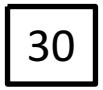
c = 20

c

12117

y = a

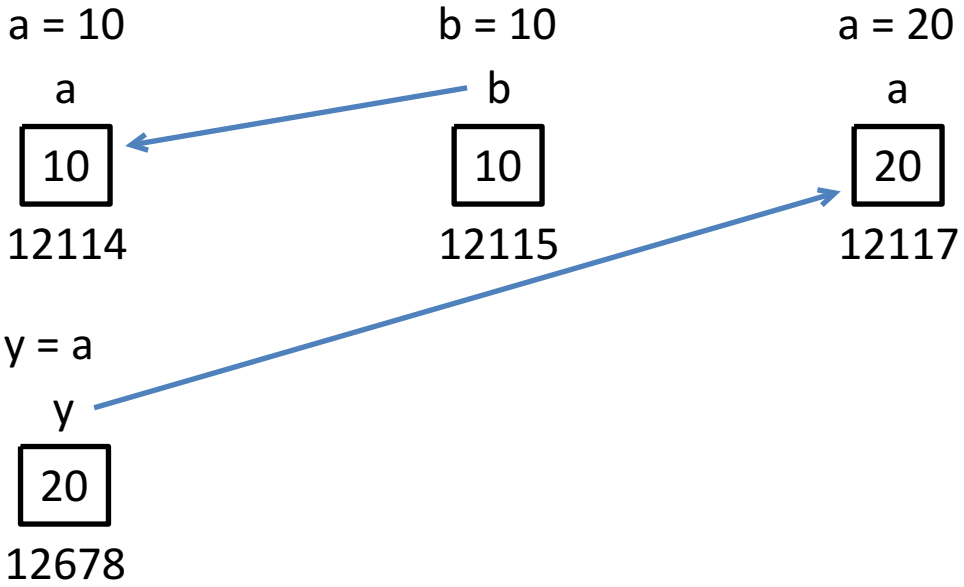
y



12678

Variable

In Python, a variable is considered as tag that is tied to some value. Python considers value as objects.



Variable

In Python, a variable is considered as tag that is tied to some value. Python considers value as objects.

Since value 10 becomes unreferenced object, it is removed by garbage collector.

Rules

- Every variable name should start with alphabets or underscore (_).
- No spaces are allowed in variable declaration.
- Except underscore (_) no other special symbol are allowed in the middle of the variable declaration
- A variable is written with a combination of letters, numbers and special characters _ (underscore)
- No Reserved keyword

Examples

Do

- A
- a
- name
- name15
- _city
- Full_name
- FullName

Don’t

- and
- 15name
- $city
- Full$Name
- Full Name

DataType

Datatype represents the type of data stored into a variable or memory.

Type of Data type :-

- Built-in Data type
- User Defined Data type

Built-in Datatype

These datatypes are provided by Python Language.

Following are the built-in data type:-

- None Type
- Numeric Types
- Sequences
- Sets
- Mappings

User Defined Data type

- Array
- Class
- Module

None Type

None datatype represents an object that doesn’t contain any value.

Numeric Type / Number

Following are the Numeric Data type:-

Int

Float

Complex

Numeric Type / Number

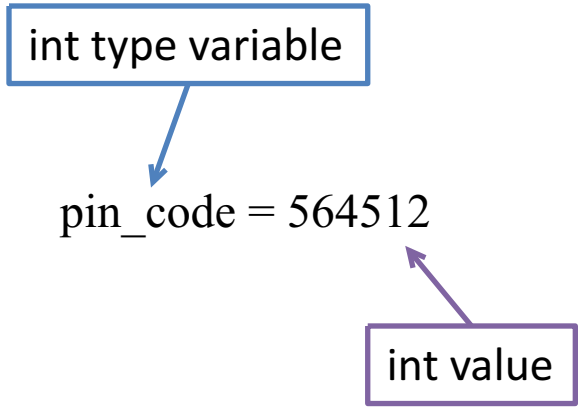
Int – The int datatype represents an integer number. An integer number without any decimal point or fraction part. In Python, It is possible to store very large integer number as there is no limit for the size of an int datatype.

Ex:-

20, 10, -50, -1002

y = 10

pin_code = 564512



Numeric Type / Number

Float – The float data type represents floating point numbers. A floating point number is a number that contains a decimal point.

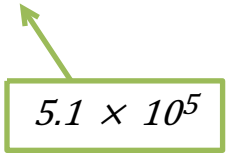
Ex:-

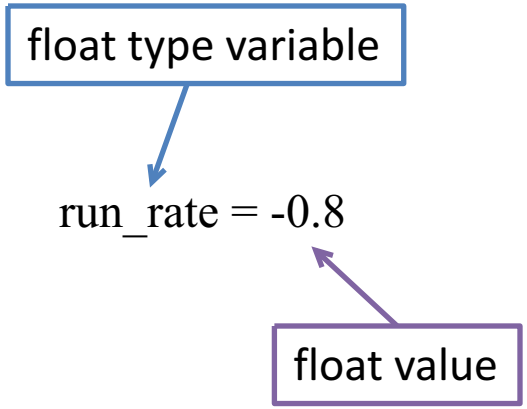
25.56, 10.5, -45.69, -0.8

price = 25.56

run_rate = -0.8

value = 5.1e5





5.1e5

It's scientific notation where e or E represents exponentiation which represents the power of 10

Numeric Type / Number

Complex – A complex number is a number that is written in the form of a + bj or a + bJ. Where,

a = Real Part of the number

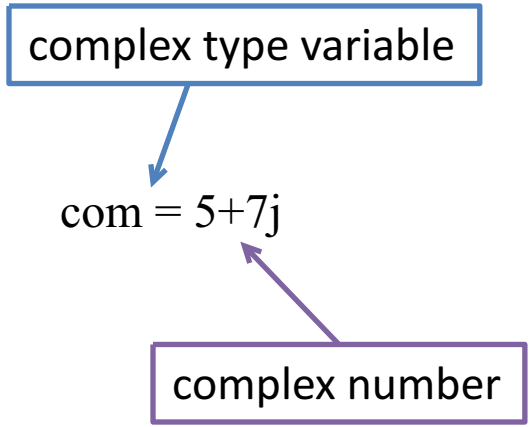
b = Imaginary part of the number

j or J = Square root value of -1

a and b may contain integer or float number.

Ex:- 5+7j, 0.8+2j

com = 5+7j



Bool type

The bool datatype represents boolean value True or False. Python internally represents True as 1 and False as 0.

Ex:- True, False

True + True = 2

True – False = 1

Sequence Type

Following are sequence type:-

String

List

Tuple

Range

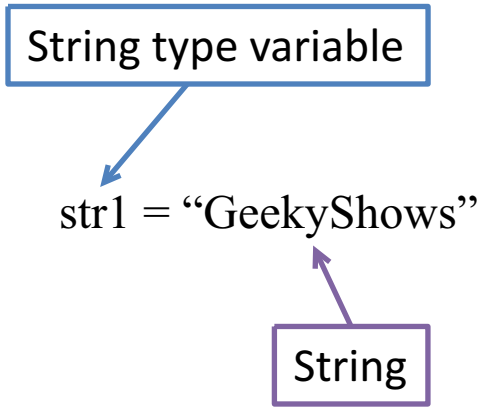
Sequence Type

String – String represents group of characters. Strings are enclosed in double quotes or single quotes.

Ex:- “Hello”, “GeekyShows”, ‘Rahul’

str1 = “GeekyShows”

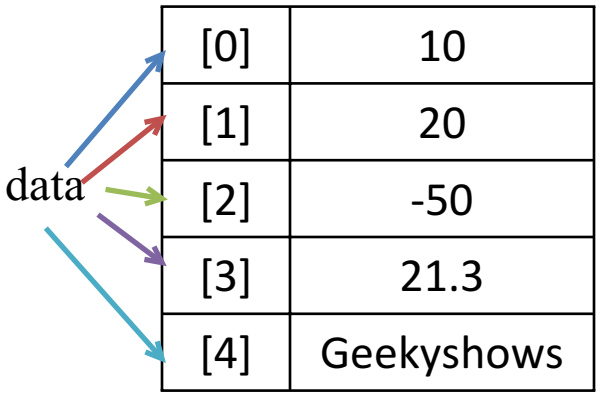
str1 = ‘GeekyShows’

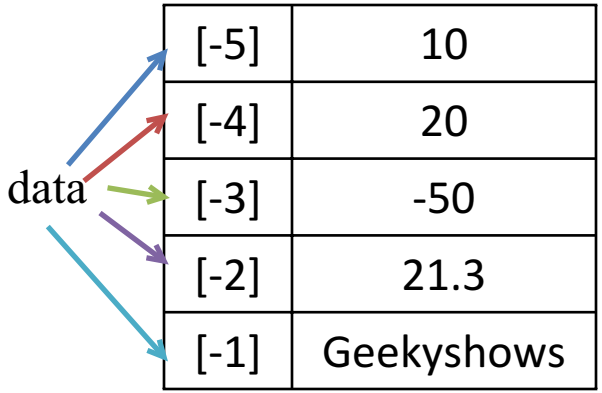


Sequence Type

List – A list represents a group of elements. A list can store different types of elements which can be modified. Lists are dynamic which means size is not fixed. Lists are represented using square bracket [ ].

Ex:- data = [10, 20, -50, 21.3, 'Geekyshows']



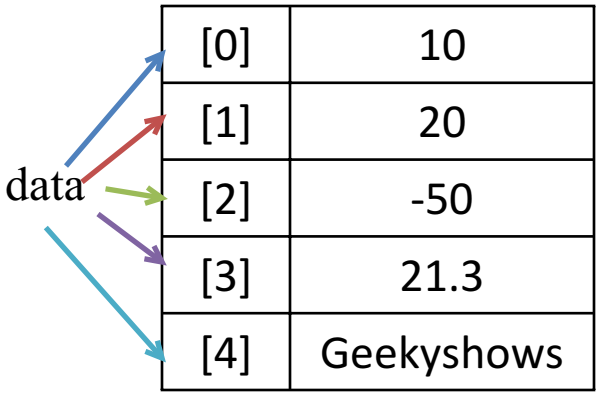


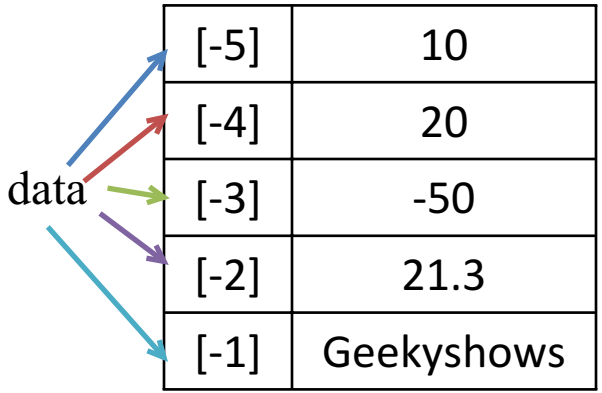
data[1] = 40

Sequence Type

Tuple – A tuple contains a group of elements which can be different types. It is similar to List but Tuples are read-only which means we can not modify it's element. Tuples are represented using parentheses ( ).

Ex:- data = (10, 20, -50, 21.3, 'Geekyshows')





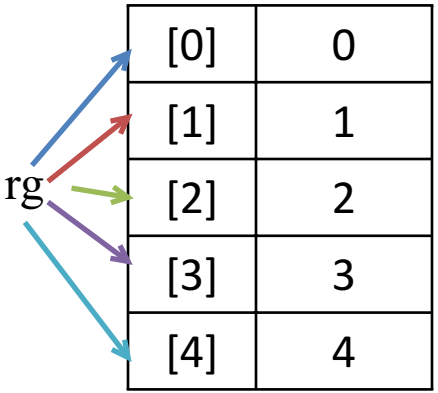
data[1] = 40

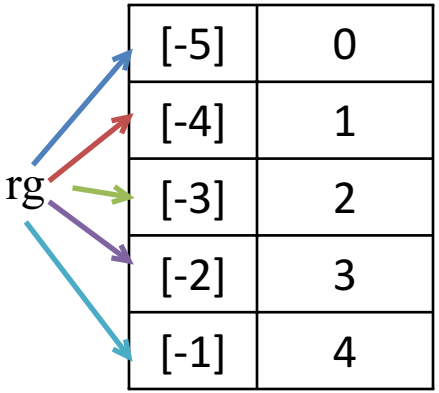
Sequence Type

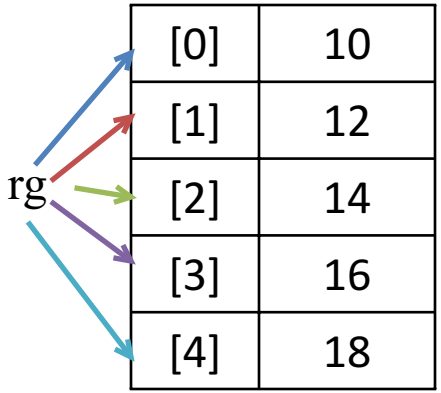
Range – Range represents a sequence of numbers. The numbers in the range are not modifiable.

Ex:- rg = range(5) 0 1 2 3 4

rg = range(10, 20, 2) 10 12 14 16 18







Set Type

A set is an unordered collection of elements much like a set in mathematics.

The order of elements is not maintained in the sets. It means the elements may not appear in the same order as they are entered into the set.

A set does not accept duplicate elements.

Sets are unordered so we can not access its element using index.

data[0] = 10

Sets are represented using curly brackets { }.

Ex:-

data = {10, 20, 30, “GeekyShows”, “Raj”, 40}

data = {10, 20, 30, “GeekyShows”, “Raj”, 40, 10, 20}

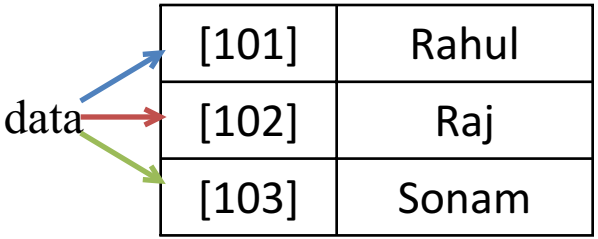
Mapping Type/ dict / Dictionary

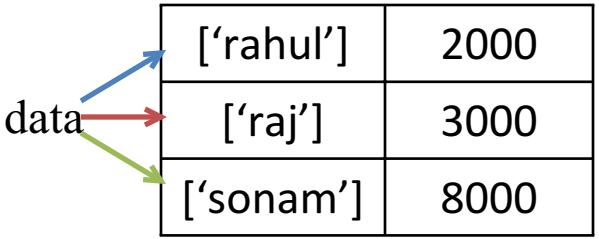
A map represents a group of elements in the form of key value pairs.

Ex:-

data = {101: ‘Rahul’, 102: ‘Raj’, 103: ‘Sonam’ }

data = {‘rahul’:2000, ‘raj’:3000, ‘sonam’:8000, }





Character

There is no concept of char data type in Python to represent individual character.

Operators

An operator is a symbol that performs an operation.

- Arithmetic Operators
- Relational Operators / Comparison Operators
- Logical Operators
- Assignment Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Arithmetic Operators

Arithmetic Operators are used to perform basic arithmetic operations like addition, subtraction, division etc.

Relational/ Comparison Operators

Relational operators are used to compare the value of operands (expressions) to produce a logical value. A logical value is either True or False.

Operators	Meaning	Example	Result
<	Less than	5<2	False
>	Greater than	5>2	True
<=	Less than or equal to	5<=2	False
>=	Greater than or equal to	5>=2	True
==	Equal to	5==2	False
!=	Not equal to	5!=2	True

Logical Operators

Logical operators are used to connect more relational operations to form a complex expression called logical expression. A value obtained by evaluating a logical expression is always logical, i.e. either True or False.

Operator	Meaning	Example	Result
and	Logical and	(5<2) and (5>3)	False
or	Logical or	(5<2) or (5>3)	True
not	Logical not	not (5<2)	True

and

Operand 1	Operand 2	Result
True	True	True
True	False	False
False	True	False
False	False	False
True	Expression	Expression
False	Expression	False

True and Expression1 and Expression2 = Expression2
False and Expression1 and Expression2 = False

or

Operand 1	Operand 2	Result
True	True	True
True	False	True
False	True	True
False	False	False
True	Expression	True
False	Expression	Expression

True or Expression1 or Expression2 = True
False or Expression1 or Expression2 = Expression1

not

Operand	Result
False	True
True	False

Assignment Operators

Assignment operators are used to perform arithmetic operations while assigning a value to a variable.

Bitwise Operators

Bitwise operators are used to perform operations at binary digit level. These operators are not commonly used and are used only in special applications where optimized use of storage is required.

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR / Bitwise XOR
~	Bitwise inversion (one's complement)
<<	Shifts the bits to left / Bitwise Left Shift
>>	Shifts the bits to right / Bitwise Right Shift

Bitwise AND &

Operand 1	Operand 2	Result (operand1 & operand2)
True 1	True 1	True 1
True 1	False 0	False 0
False 0	True 1	False 0
False 0	False 0	False 0

a = 10 0 0 0 0 1 0 1 0
b = 15 0 0 0 0 1 1 1 1

Bitwise OR |

Operand 1	Operand 2	Result (operand1 | operand2)
True 1	True 1	True 1
True 1	False 0	True 1
False 0	True 1	True 1
False 0	False 0	False 0

a = 10 0 0 0 0 1 0 1 0

b = 15 0 0 0 0 1 1 1 1

Bitwise XOR ^

Operand 1	Operand 2	Result (operand1 ^ operand2)
True 1	True 1	False 0
True 1	False 0	True 1
False 0	True 1	True 1
False 0	False 0	False 0

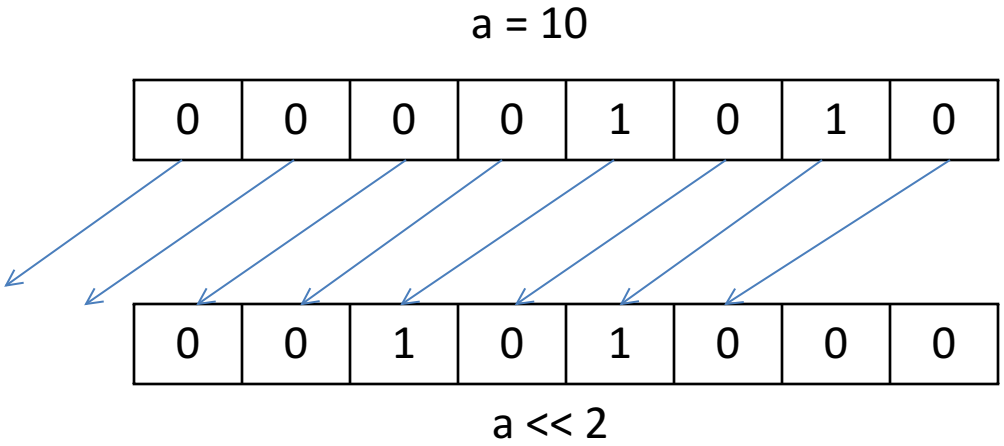
a = 10 0 0 0 0 1 0 1 0
b = 15 0 0 0 0 1 1 1 1

Bitwise NOT ~

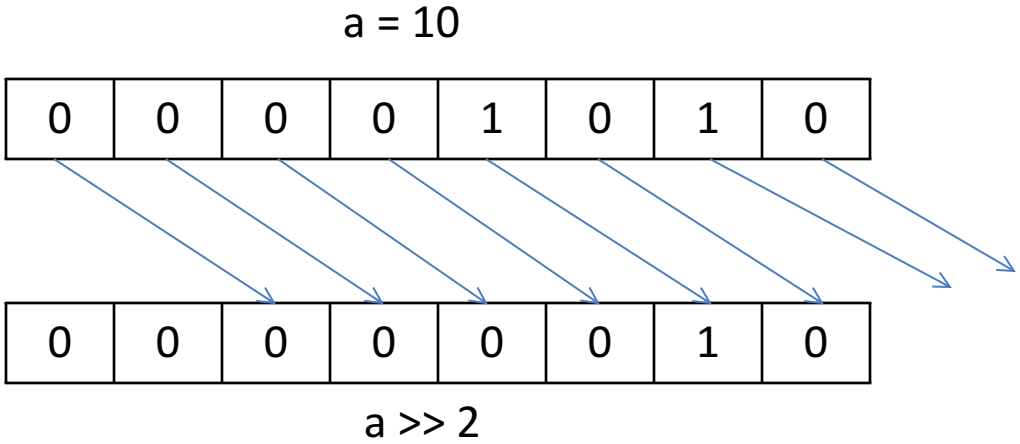
Operand	Result (~ operand)
True 1	False 0
False 0	True 1

a = 10 0 0 0 0 1 0 1 0

Bitwise Left Shift <<



Bitwise Right Shift >>



Membership Operators

The membership operators are useful to test for membership in a sequence such as string, lists, tuples and dictionaries.

There are two type of Membership operator:-

- in
- not in

in

This operators is used to find an element in the specified sequence.

It returns True if element is found in the specified sequence else it returns False.

Ex:-

st1 = “Welcome to geekyshows”

“to” in st1 True

st2 = “Welcome top geekyshows”

“to” in st2 True

st3 = “Welcome to geekyshows”

“subs” in st3 False

not in

This operators works in reverse manner for in operator.

It returns True if element is not found in the specified sequence and if element is found, then it returns False.

Ex:-

st1 = “Welcome to geekyshows”

“subs” not in st1 True

st2 = “Welcome to geekyshows”

“to” not in st2 False

st3 = “Welcome top geekyshows”

“to” not in st3 False

Identity Operators

The identity operators compare the memory locations of two objects. Hence, it is possible to know whether two objects are same or not.

There are two types of Identity operator:-

- is
- is not

is

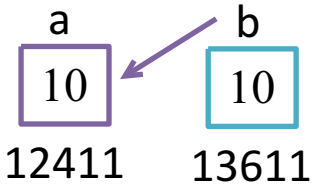
This operator is used to compare whether two objects are same or not.

It returns True if memory location of two objects are same else it returns False.

Ex:-

a = 10

b = 10



a is b True

a = 10

b = '10'

a is b False

is not

This operator works in reverse manner for *is* operator.

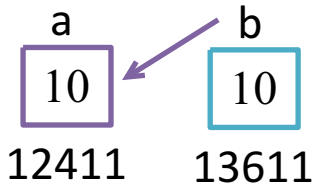
It returns True if memory location of two objects are not same and if they are same it returns False.

Ex:-

a = 10

b = 10

a is not b False



a = 10

b = '10'

a is not b True

Operator Precedence and Associativity

The computer scans an expression which contains the operators from left to right and performs only one operation at a time. The expression will be scanned many times to produce the result. The order in which various operations are performed is known as hierarchy of operations or operator precedence. Some of the operators of the same level of precedence are evaluated from left to right or right to left. This is referred to associativity.

Order	Operator	Meaning
1	**( )**	Parentheses
2	**	Exponentiation
3	+, -, ~	Unary Plus, Unary Minus, Bitwise Not
4	*, /, //, %	Multiplication, Division, Floor Division, Modulus
5	+, -	Addition, Subtraction
6	<<, >>	Bitwise Left Shift, Bitwise Right Shift
7	&	Bitwise AND
8	^	Bitwise XOR
9	>, >=, <, <=, ==, !=	Relational Operators
10	=, %=, /=, //=, -=, +=, *=, **=	Assignment Operators
11	is, is not	Identity Operators
12	in, not in	Membership Operators
13	not	Logical NOT
14	or	Logical OR
15	and	Logical AND

- Parentheses
- Exponentiation
- Multiplication, Division, Modulus and Floor Division
- Addition and Subtraction
- Assignment

```
value = (1+1)*2**4//3+4-1
        2*2**4//3+4-1
        2*16//3+4-1
        32//3+4-1
        10+4-1
        14-1
        13
```

Type Conversion

Converting one data type into another data type is called *Type Conversion*.

Type of *Type Conversion*:-

- Implicit Type Conversion
- Explicit Type Conversion

Implicit Type Conversion

In the Implicit type conversion, python automatically converts one data type into another data type.

Ex:-

```
a = 5
b = 2
value = a / b
print(value)
print(type(value))
```

Explicit Type Conversion

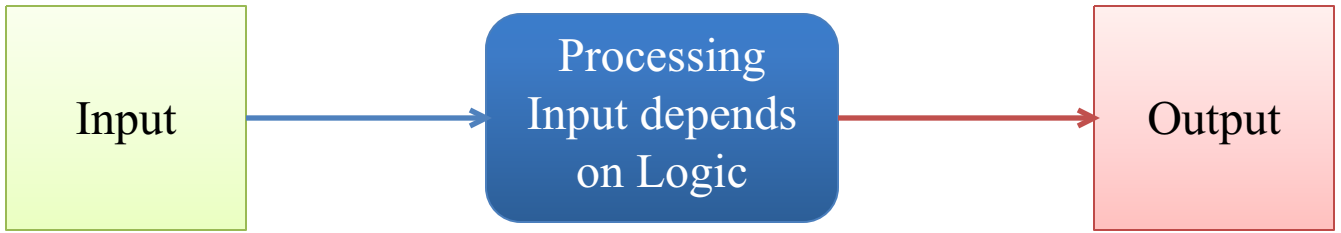
In the Cast/Explicit Type Conversion, Programmer converts one data type into another data type.

- int (n)
- float (n)
- complex (n)
- complex (x, y) where x is real part and y is imaginary part
- str (n)
- list(n)
- tuple(n)
- bin (n)
- oct (n)
- hex (n)

Input and Output

Input - The data given to the computer is called input.

Output – The results returned by the computer are called output.



Keyboard
Mouse
Database

Output Statements

print ( ) Function - The print() function is used to print the specified message to the output screen/device. The message can be a string, or any other object.

Syntax:- print(objects, sep='character', end='character', file=sys.stdout, flush=False)

sep - Separate the objects by given character. Character can be any string. Default is ' ' or can write none.

end – It indicates ending character for the line. Default is '\n' or can write none.

file - An object with a write method. Default is sys.stdout or can write none.

flush - A Boolean, specifying if the output is flushed (True) or buffered (False). Default is False

Output Statements

print ( ) – This function is used to display a blank line.

print(“string”) - When a string is passed to the function, the string is displayed as it is.

Ex:-

```
print(“Welcome to Geeky Shows”)
print(‘Welcome to Geeky Shows’)
print(“Like”, “Share”, “Subscribe”)
print(10)

print(“Welcome”)
print(“to”)
print(“Geeky Shows”)
```

Output Statements

print(object) - We can pass objects like list, tuples and dictionaries to display the elements of those objects.

Ex:-

data = [10, 20, -50, 21.3, ‘Geekyshows’]

print(data)

Output Statements

print(“string” sep=‘’) – It separates string with given sep character. Character can be any string. Default is ‘ ’ or can write none.

Ex:-

print(“Like”, “Share”, “Subscribe”, sep=‘’)

print(“Like”, “Share”, “Subscribe”, sep=‘***’)

Output Statements

print(“string” end=‘’) – When ending character is passed. It prints given character at the end. Default is ‘\n’ or can write none.

Ex:-

```
print(“Welcome”, end=‘\n’)

print(“Welcome”, end=‘’)
print(“to”, end=‘’)
print(“GeekyShows”)

print(“Welcome”, end=‘\t’)
print(“to”, end=‘\t’)
print(“GeekyShows”)
```

Output Statements

print(variable list) – This is used to display the value of a variable or a list of variable.

Ex:-

```
a = 10
print(a)
```

Output: 10

```
x = 20
y = 30
print(x, y)
```

Output: 20 30

List of variable's value in the output screen, are separated by a space by default. we can change this by sep

```
print(x, y, sep=',')
```

Output: 20, 30

Output Statements

print(“String”, variable list) – This is used to display the string along with variable.

Ex:-

m = 40

print(“Value: ”, m)

Output: Value: 40

name = “Rahul”

age = 62

print(“My Name is ”, name, “and My age is”, age)

Output: My Name is Rahul and My age is 62

Input Statements

input( ) – This function is used to accept input from keyboard.

This function will stop the program flow until the user gives an input and end the input with the return key.

Whatever user gives as input, input function convert it into a string. If user enters an integer value still input() function convert it into a string.

So if you need an integer you have to use type conversion.

Syntax:- input([prompt])

prompt is a string or message, representing a default message before input. It is optional

Ex:-

name = input( )

name = input(“Your Name: ”)

mobile = input(“Enter Your Mobile Number: ”)

Input Statements

Whatever user gives as input, input function convert it into a string. If user enters an integer value still input() function convert it into a string.

So if you need an integer you have to use type conversion.

Ex:-

```
mobile = input(“Enter Your Mobile Number: ”)
mb = int(mobile)

mobile = int ( input (“Enter Your Mobile Number: ”) )
price = float ( input (“Total Price: ”) )
mobile = complex ( input (“Enter Complex Number: ”) )
```

Escape Sequence

Escape sequences – Escape sequences are control character used to move the cursor and print characters such as ‘, “. \ and so on.

Escape Sequence	Meaning
\\	Backslash
\’	Single Quote
\”	Double Quote

Escape Sequence	Meaning
\a	Bell
\b	Backspace
\f	Formfeed
\n	NewLine
\r	Carriage Return
\t	Horizontal Tab
\v	Vertical Tab
\newline	Backslash and NewLine Ignored

Comment

Comments are non-executable statements. A Comment is used to describe the feature of a program. Comment helps to understand our program, not only ourselves but also other programmer.

There are two type of programs:-

- Single Line Comment
- Multi line Comment

Single Line Comment

These comments start with a hash symbol (#).

Ex:-

I am single Line Comment

This is my first Python Program

Adding two numbers

Multi Line Comment

There is no concept of multi line comment in python but we can create string starting and ending with triple double quotes (""") or triple single quotes (''') which can be used as block of comments.

Since strings are not assigned to any variable, then they are removed from memory by the garbage collector and hence these can be used as comments.

It is not recommended to use triple double quotes or triple single quotes for writing comments as it internally occupy memory and would waste time of the interpreter since the interpreter has to check them.

Ex:-

```
"""
Comment Line 1
Comment Line 2
Comment Line 3
"""
```

```
'''
Comment Line 1
Comment Line 2
Comment Line 3
'''
```

If Statement

It is used to execute an instruction or block of instructions only if a condition is fulfilled.

Syntax: -

```
if (condition):
    statement
Rest of the Code

if (condition):
    statement1
    statement2
Rest of the Code
```

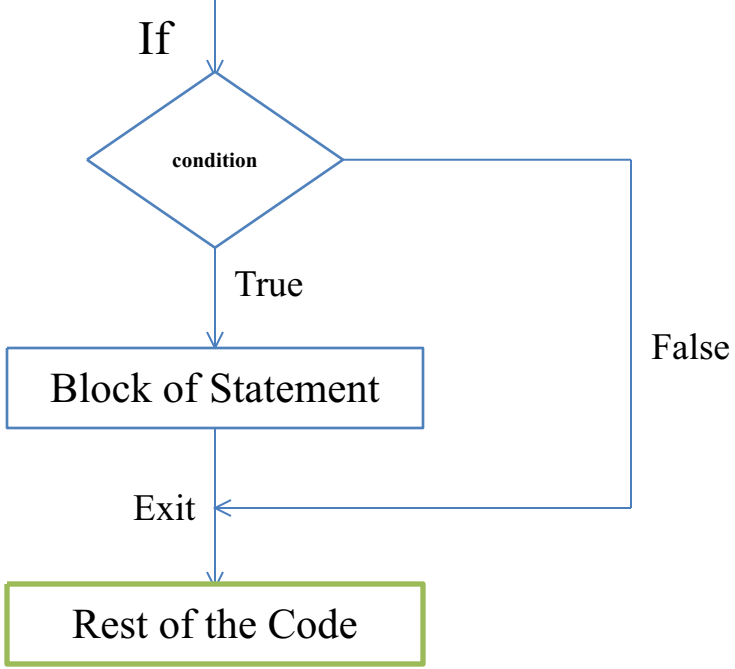
First the condition is tested, If the condition is True

then the statements given after colon (:) are executed

Block of statement/ Group of statements/ Suite

If there is single statement it can be written in one line.

Ex:- if (condition): Statement


If
condition
True
False
Block of Statement
Exit
Rest of the Code

Nested If Statement

Syntax :

```
if (condition):
    block of statements
    if(condition):
        block of statements
    if(condition):
        block of statements
if(condition):
    block of statements
Rest of the code
```

If Statement with Logical Operator

```
if ( (condition1) and (condition2) ):
        Statement
Rest of the Code

if ( (condition1) and (condition2) ):
        Block of Statements
Rest of the Code

if ( (condition1) or (condition2) ):
        Statement
Rest of the Code
```

Indentation

Indentation refers to spaces that are used in the beginning of a statement. By default python puts 4 spaces but it can be changed by programmers.

```
if (condition):
    Statement
    if(condition):
        statement 1
        statement 2
    if(condition):
        Statement
if(condition):
    Statement 1
    Statement 2
Rest of the code
```

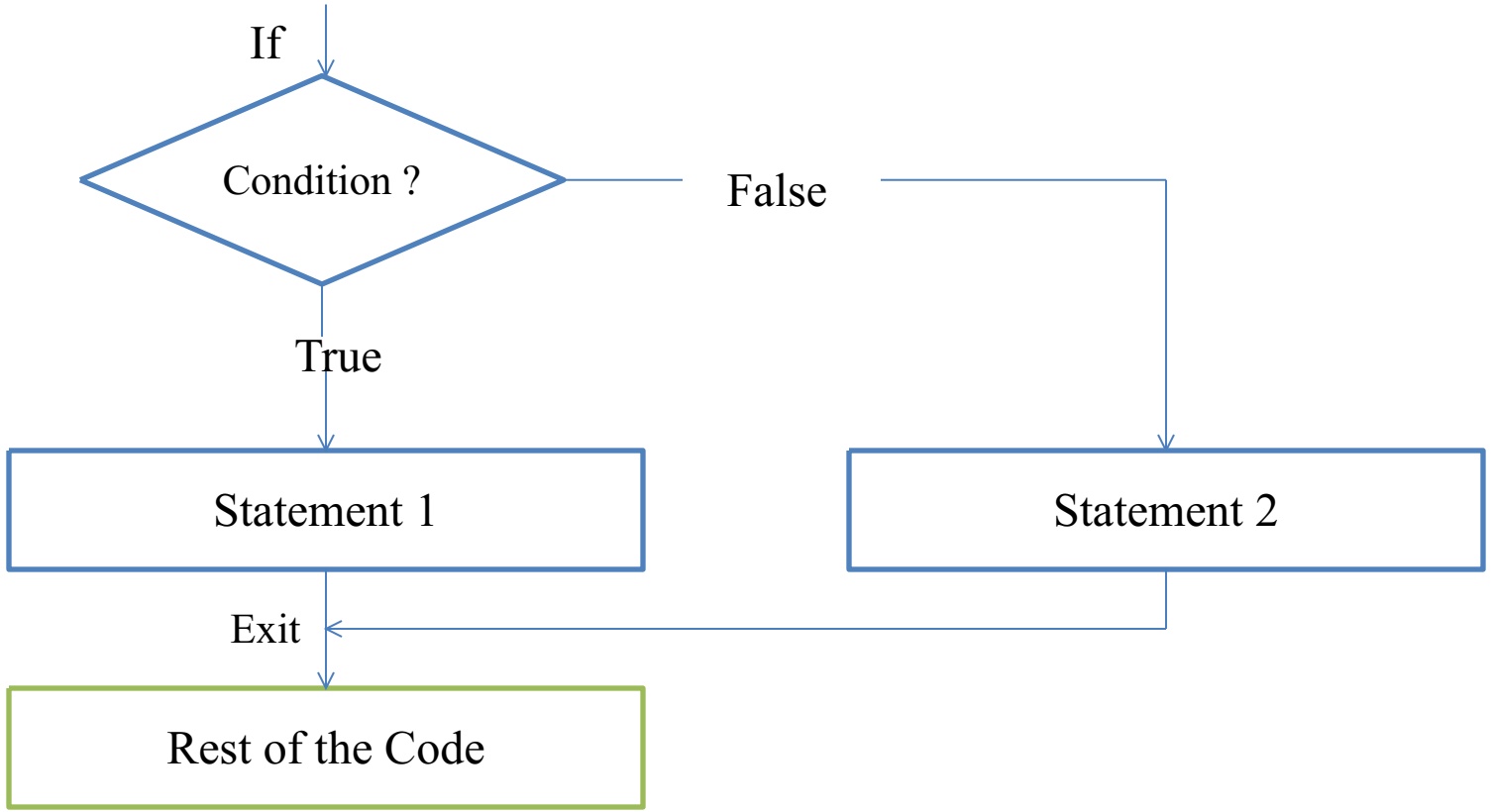
If Else Statement

if… else statement is used when a different sequence of instructions is to be executed depending on the logical value(True/False) of the condition evaluated.

Syntax: -

```
if(condition):
    Statement 1
else:
    Statement 2
Rest of the Code
```

```
if(condition):
    Statement 1
    Statement 2
else:
    Statement 3
    Statement 4
Rest of the Code
```


If
Condition ?
False
True
Statement 1
Statement 2
Exit
Rest of the Code

Nested If Else Statement

In nested if…. else statement, an entire if… else construct is written within either the body of the if statement or the body of an else statement.

Nested If Else Statement

```
if(condition_1):
        if(condition_2):
                Statement 1
         else:
                Statement 2
else:
        Statement 3
Rest of the Code
```

```
if(condition_1):
        if(condition_2):
                Statement 1
         else:
                Statement 2
else:
         if(condition_3):
                Statement 3
          else:
                 Statement 4
Rest of the Code
```

if elif Statement

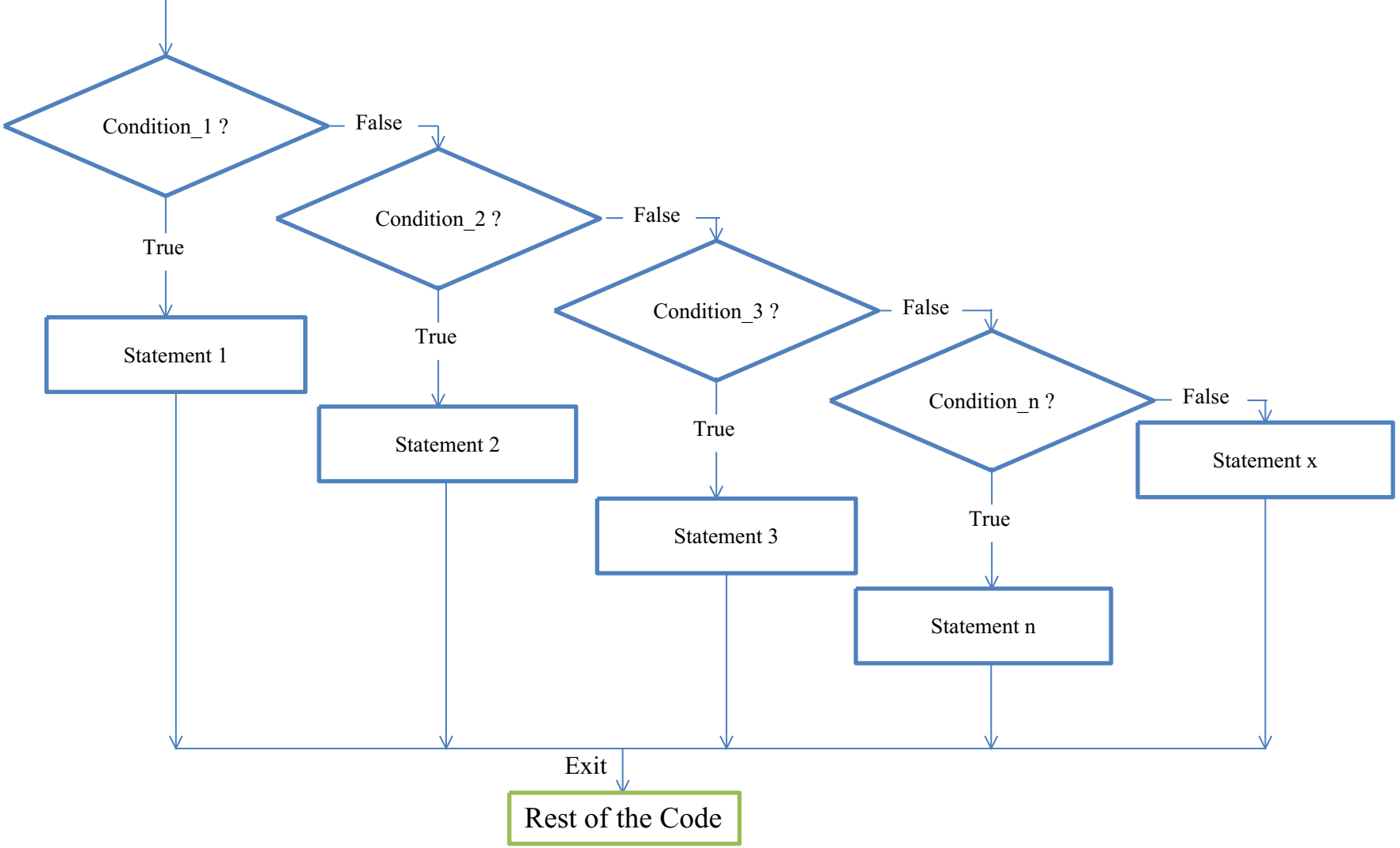
To show a multi-way decision based on several conditions, we use if elif statement.

```
if (condition_1):
        Statement 1
elif (condition_2):
        Statement 2
elif (condition_3):
        Statement 3
elif (condition_n):
       Statement n
Rest of the Code
```

if elif else Statement

To show a multi-way decision based on several conditions, we use if elif else statement.

```
if (condition_1):
        Statement 1
elif (condition_2):
        Statement 2
elif (condition_3):
        Statement 3
elif (condition_n):
        Statement n
else:
        Statements x
Rest of the Code
```


Condition_1 ?
False
Condition_2 ?
False
Condition_3 ?
False
Condition_n ?
False
Statement x
True
Statement 1
True
Statement 2
True
Statement 3
True
Statement n
Exit
Rest of the Code

Loop Control Statements

Loop control statements are used when a section of code may either be executed a fixed number of times, or while some condition is true.

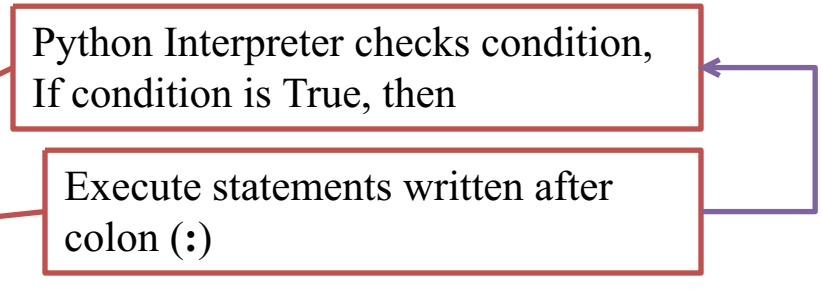
- While
- For

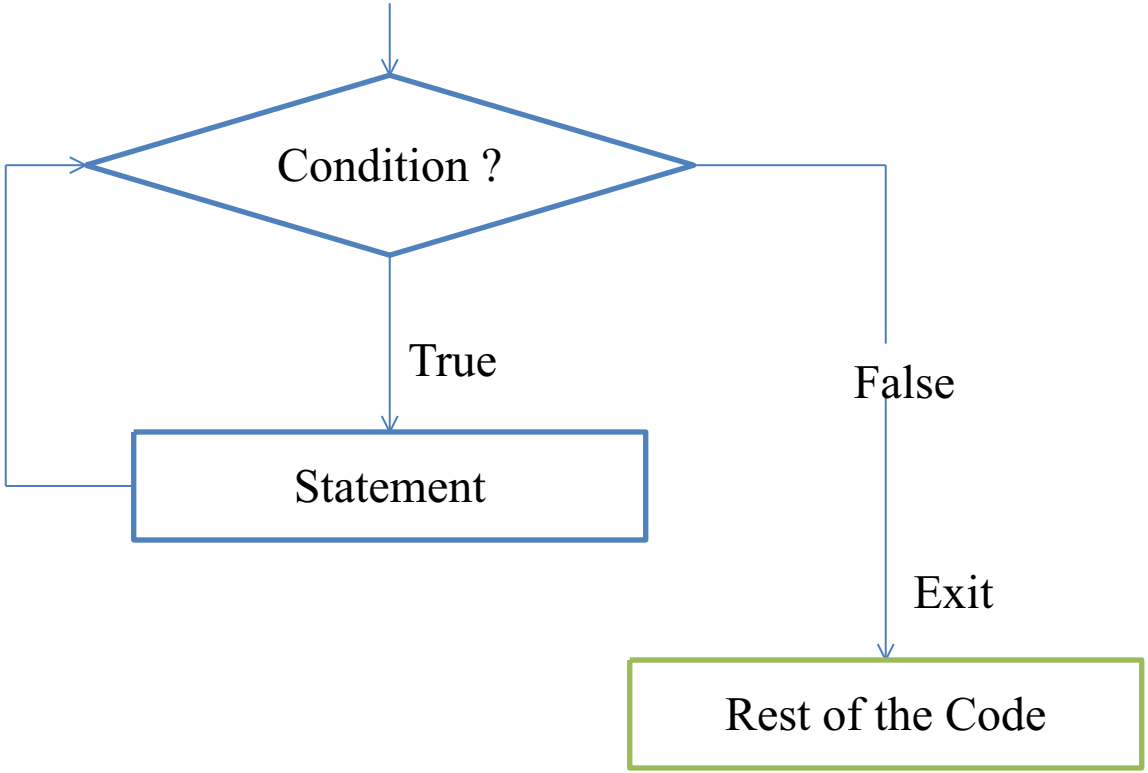
while Loop

The while loop keeps repeating an action until an associated condition returns false.

Syntax:

```
while (condition):
        Statement
Rest of the Code
```



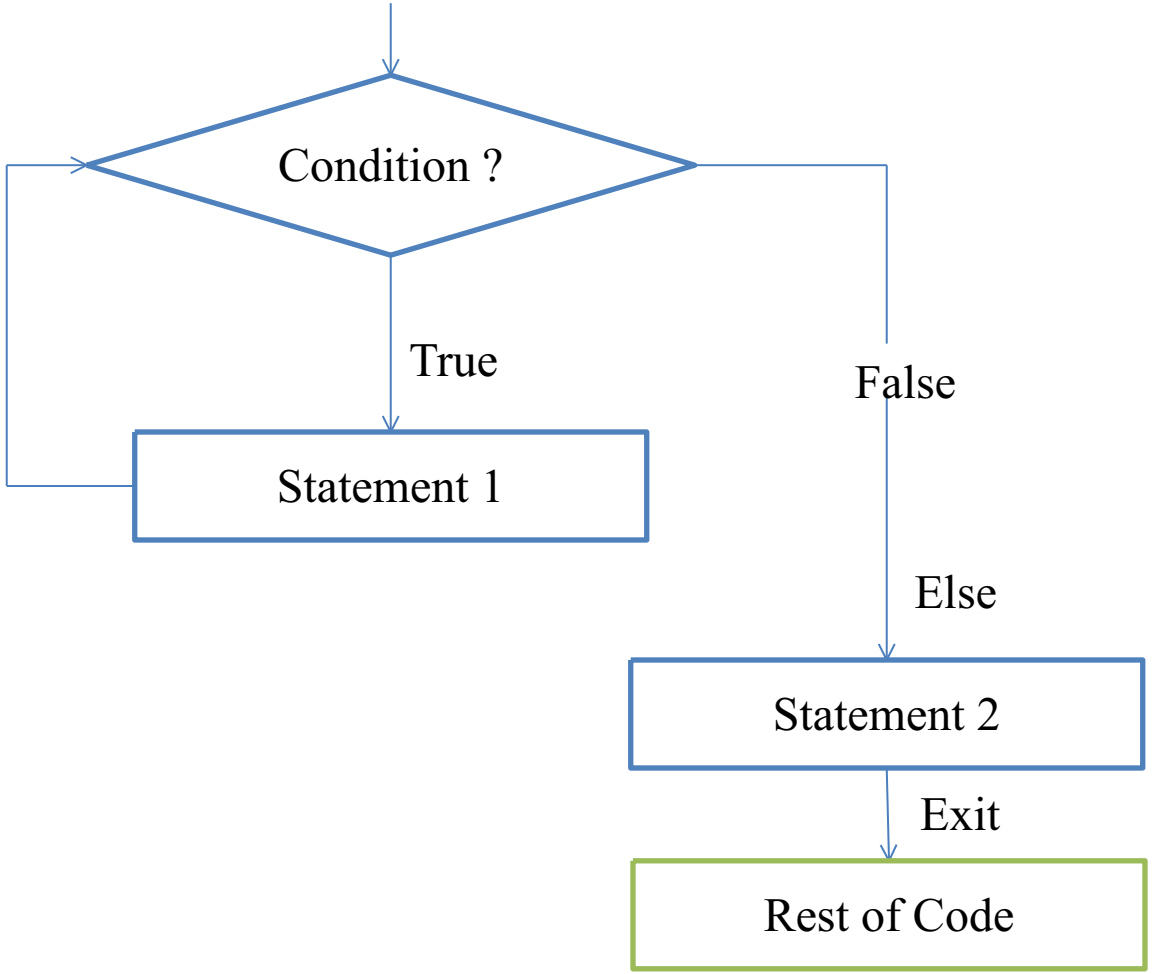

Condition ?
True
False
Statement
Exit
Rest of the Code

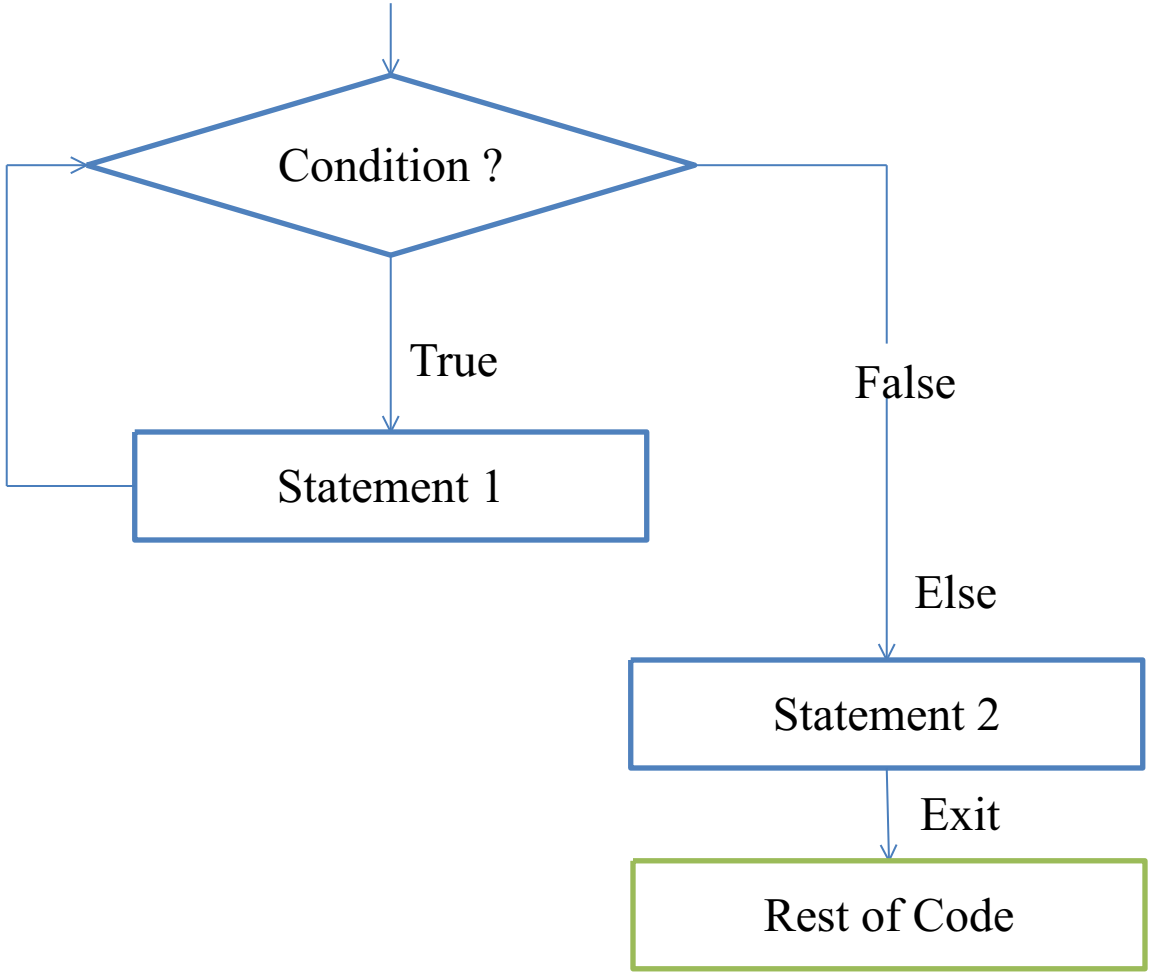
while Loop with else

This repeatedly tests the condition and, if it is True, executes the Statement 1; if the condition is False (which may be the first time it is tested) the Statement 2 of the else clause, is executed and the loop terminates. The else suite will be always executed irrespective of the statements in the loop are executed or not.

Syntax:

```
while (condition):
        Statement 1
else:
        Statement 2
Rest of the Code
```


Condition ?
True
False
Statement 1
Else
Statement 2
Exit
Rest of Code


Condition ?
True
False
Statement 1
Else
Statement 2
Exit
Rest of Code

Infinite while Loop

Syntax:

```
while (True):
    Statement
Rest of the Code
```

```
while (True):
    Statement
    if(condition):
        break
Rest of the Code
```

Nested While Loop

```
while(condition):
        Statements
        while(condition):
                Statements
        Statements
Rest of Code
```

range( ) Function

range() function is used to generate a sequence of integers starting from 0 by default, and increments by 1 by default, till j-1.

Syntax:-

range(start, stop, stepsize)

Start – Starting position. If we do not mention start by default it’s 0

*Stop – Ending position. The range of integers stops one element prior to stop. If stop is j then it will stop at exact j-1

Stepsize – Increment by stepsize. If we do not mention start by default it’s 1

range( ) Function

Syntax:- range(j)	0, 1, 2, 3, 4,……..., j-1
Ex:- range(10)	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Syntax:- range(i, j)	i, i+1, i+2. i+3,….., j-1
Ex:- range(1, 10)	1, 2, 3, 4, 5, 6, 7, 8, 9
Syntax: - range(i, j, k)	i, i+k, i+2k, i+3k, i+4k,….., j-1
range(1, 10, 2)	1, 3, 5, 7, 9
range(-1, -10, -2)	-1 -3 -5 -7 -9
range(10, 0, -1)	10 9 8 7 6 5 4 3 2 1

Rules:-

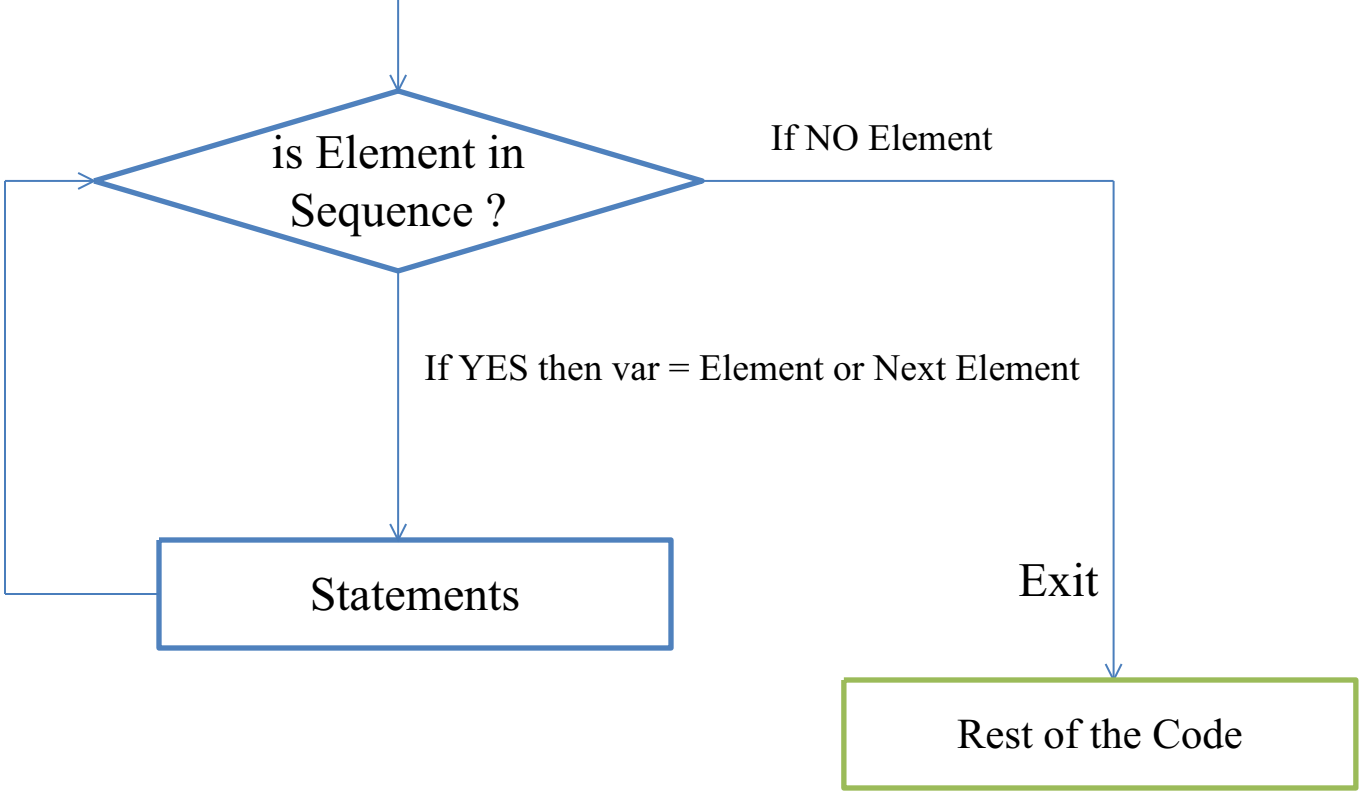
- All argument must be integers, whether its positive or negative
- You can not pass a string or float number or any other type in a start, stop and stepsize.
- The stepsize value should not be zero.

for Loop

The for loop is useful to iterate over the elements of sequence such as string, list, tuple etc.

Syntax:

```
for var in sequence:
        Statements
Rest of the Code
```


is Element in Sequence ?
If NO Element
If YES then var = Element or Next Element
Statements
Exit
Rest of the Code

for Loop with Range

```
a = range(5)
for i in a :
        print(i)

for i in range(5) :
        print(i)
```

for Loop with else

The for loop is useful to iterate over the elements of sequence such as string, list, tuple etc. The else suite will be always executed irrespective of the statements in the loop are executed or not.

Syntax:

```
for var in sequence:
        Statements
else:
        Statements
Rest of the Code
```

Nested for loop

For loop inside another for loop is known as nested for loop.

```
for i in range(n):
        for j in range(y):
            Statements
        Statements
Rest of the Code
```

Break Statement

Break statement is used to jump out of loop to process the next statement in the program.

```
while condition:
        if(condition):
                break
Rest of Code
```

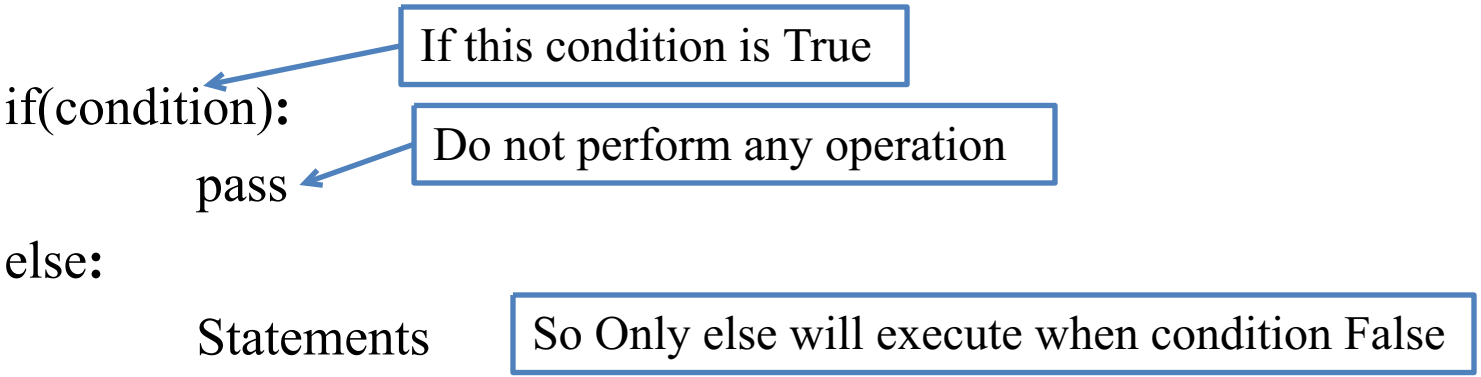
Continue Statement

Continue statement is used in a loop to go back to the beginning of the loop.

```
while condition:
        if(condition):
                continue
Rest of Code
```

Pass Statement

Pass Statement is used to do nothing. It can be used inside a loop or if statement to represent no operation. Pass is useful when we need statement syntactically correct but we do not want to do any operation.



Pass Statement

```
while condition:
        if(condition):
                pass
        Statements
Rest of Code
```

Array

In Python, Array is an object that provide a mechanism for storing several data items with only one identifier, thereby simplifying the task of data management. Array is beneficial if you need to store group of elements of same datatype.

In Python, Arrays can increase or decrease their size dynamically.

Group of Students

Group of Employees

Group of Dogs

Group of Integer- 10, 2, 40, 5, 6

Group of Float – 15.4, 25.4, 6.5

- Arrays can store only one type of data.
- In Python, The size of array is not fixed. Array can increase or decrease their size dynamically.
- Array and List are not same.
- Array uses less memory than List.

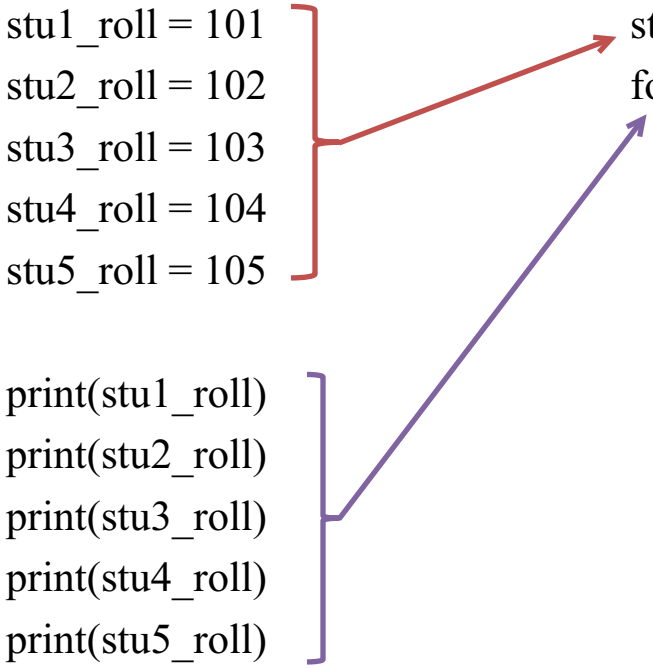
Why we need Array ?

101, 102, 103, 104, 105

```
stu1_roll = 101
stu2_roll = 102
stu3_roll = 103
stu4_roll = 104
stu5_roll = 105

print(stu1_roll)
print(stu2_roll)
print(stu3_roll)
print(stu4_roll)
print(stu5_roll)
```

101, 102, 103, 104, 105

```
stu_roll = array('i', [101,102,103,104,105])
for element in stu_roll:
        print(element)
```



Type of Array

- One Dimensional Array / One D Array – Single Row Multiple Columns

  Ex:- Student's Roll Number

  [101, 102, 103, 104, 105]

- Multi-Dimensional Array / Multi D Array – Multiple Rows Multiple Columns

  Ex:- Student's Subject Marks

  [40, 60, 70, 80, 30], ← 1st students Marks

  [50, 40, 60, 30, 40] ← 2nd students Marks

Note - Python does not support Multi-Dimensional Array but we can create Multi Dimensional Array using third party packages like numpy.

One Dimensional Array

Single Row Multiple Columns

Ex:- [101, 102, 103, 104, 105]

Import Array Module

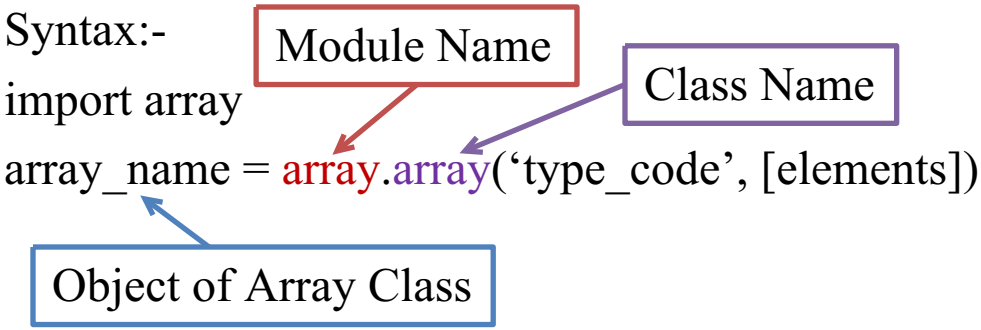
Two way to import array module:-

- import array – This will import the entire array module.

- from array import * - This will import all class, objects, variable etc from array module. Here * means All.

Creating and Initializing One-D Array

Syntax:-



Ex:-

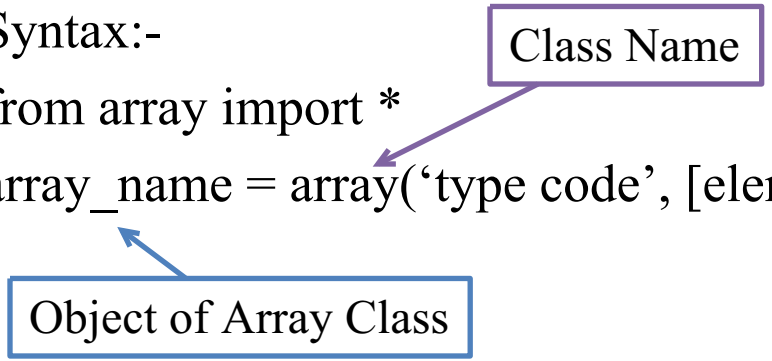
```
import array
stu_roll = array.array('i', [101, 102, 103, 104, 105])
```

Creating and Initializing One-D Array

Syntax:-

from array import *

array_name = array(‘type code’, [elements])



Ex:-

from array import *

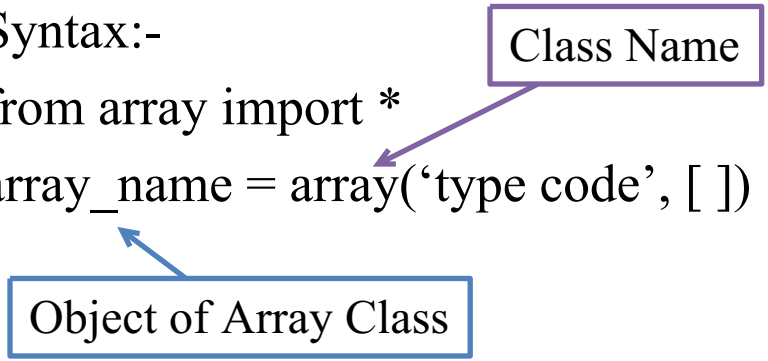
stu_roll = array(‘i’, [101, 102, 103, 104, 105])

Creating Empty One-D Array

Syntax:-

from array import *

array_name = array(‘type code’, [ ])



Ex:-

from array import *

stu_roll = array(‘i’, [ ])

Type Code

Type Code	C Type	Python Type	Size in bytes
b	Signed char	int	1
B	Unsigned char	int	1
h	Signed short	int	2
H	Unsigned short	int	2
i	Signed int	int	2
I	Unsigned int	int	2
l	Signed long	int	4
L	Unsigned long	int	4
q	Signed long long	int	8
Q	Unsigned long long	int	8
f	Float	float	4
d	Double	float	8

index

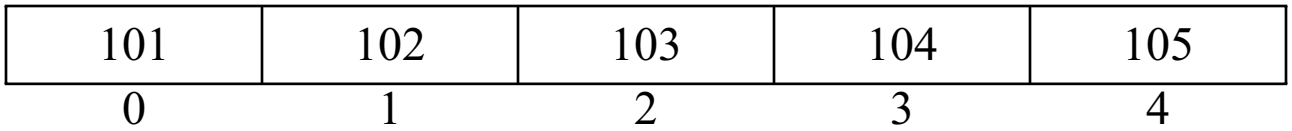
An index represents the position number of an array's element.

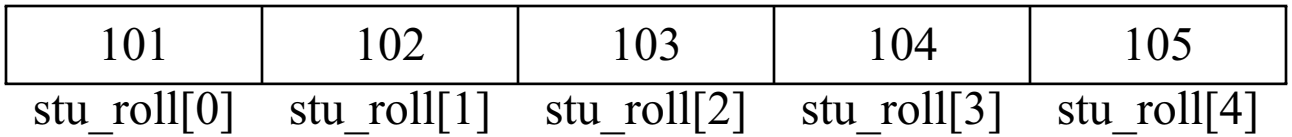
*from array import **

stu_roll = array('i', [101, 102, 103, 104, 105])

Python interpreter allocates 5 blocks of memory and stores the elements

Index always starts with 0





Accessing One-D Array Elements

*from array import **

stu_roll = array('i', [101, 102, 103, 104, 105])

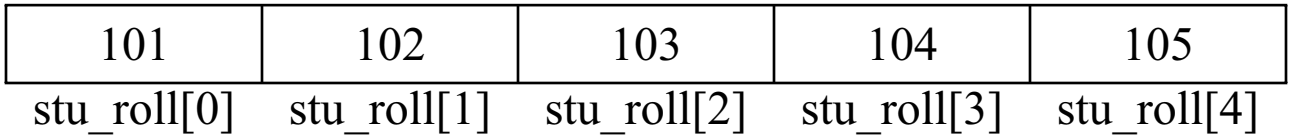
print(*stu_roll*[0])

print(*stu_roll*[1])

print(*stu_roll*[2])

print(*stu_roll*[3])

print(*stu_roll*[4])



Accessing Array using for loop

*from array import **

stu_roll = array('i', [101, 102, 103, 104, 105])

Without index

```
for element in stu_roll:
        print(element)
```

With index

```
n = len(stu_roll)
for i in range(n):
        print(stu_roll[i])
```

Accessing Array using while loop

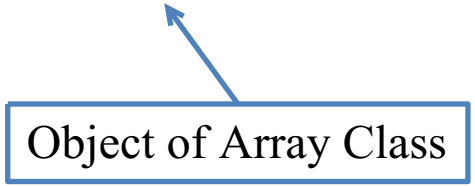
```
from array import *
stu_roll = array('i', [101, 102, 103, 104, 105])

n = len(stu_roll)
i = 0
while i < n :
        print(stu_roll[i])
        i+=1
```

append ( )

This method is used to add an element at the end of the existing array.

Syntax:-

```
array_name.append(new_element)
```



Getting User input

```
from array import *
stu_roll = array('i', [ ])
n = int(input("How many elements? "))
for i in range(n):
        stu_roll.append(int(input("Enter Element: ")))
for i in range(len(stu_roll)):
        print(stu_roll[i])
```

insert( )

This method is used to insert an element in a particular position of the existing array.

Syntax:-

```
array_name.insert(position_number, new_element)
```

pop ( )

This method is used to remove last element from the existing array.

Syntax:-

```
array_name.pop( )
```

pop (n)

This method is used to remove an element specified by position number, from the existing array and returns removed element.

Syntax:-

```
array_name.pop(position_number)
```

remove( )

This method is used to remove first occurrence of given element from the existing array. If it doesn’t found the element, shows valueError.

Syntax:-

```
array_name.remove(element)
```

index( )

This method returns position number of first occurrence of given element in the array. If it doesn’t found the element, shows valueError.

Syntax:-

array_name.index(element)

reverse ( )

This method is used to reverse the order of elements in the array.

Syntax:-

array_name.reverse( )

extend( )

This method is used to append another array or iterable object at the end of the array

Syntax:-

array_name.extend(arr)

Slicing on Arrays

Slicing on arrays can be used to retrieve a piece of the array that contains a group of elements. Slicing is useful to retrieve a range of elements.

Syntax:-

new_array_name = array_name[start:stop:stepsize]

pip

pip is the package manager for Python. Using pip we can install python packages.

How to Install pip

pip is distributed with Python which means that when you download Python from https://www.python.org/ , you automatically get pip installed on your computer.

Check if pip is already Installed

pip --version

Update pip

python -m pip install --upgrade pip

py -m pip install --upgrade pip

How to Get Help

pip help – It shows list of pip commands and their functions.

How to install Packages

pip install camelcase

How to Use Packages

import camelcase

How to Uninstall Packages

pip uninstall camelcase

numpy

In Python, Numpy is a package which contains classes, functions, variables, large library of mathematical functions etc to work with scientific calculation.

Numpy can be used to create n dimensional arrays where n is any integer. We can create 1 dimensional array, 2 dimensional array, 3 dimensional array and so on.

Numpy's array class is called ndarray. It is also known by alias name array. There is another class array in python which is different from numpy's array class.

Import numpy

There are two ways to import numpy:-

- import numpy – This will import the entire numpy module.

- from numpy import * - This will import all class, objects, variable etc from numpy package. Here * means All.

One Dimensional Array

Single Row Multiple Columns

Ex:- [101, 102, 103, 104, 105]

Ways of Creating Array in numpy

- array ( ) Function
- linspace ( ) Function
- logspace ( ) Function
- arange ( ) Function
- zeros ( ) Function
- ones ( ) Function

array ( ) Function

array ( ) Function of numpy module is used to create an array.

Syntax:-

numpy.array(object, dtype=None, copy=True, order='K', subok=False, ndmin=0)

https://www.numpy.org/devdocs/reference/generated/numpy.array.html

Creating 1D Array using array ( ) Function

Syntax:-


```
import numpy
array_name = numpy.array([elements])
```

Ex:-

```
import numpy
stu_roll = numpy.array([101, 102, 103, 104, 105])
stu_roll = numpy.array([101, 102, 103, 104, 105], int)
a = numpy.array([10.1, 5.2, 4.23, 10.4, 1.5], float)
a = numpy.array([10.1, 5.2, 4.23, 10.4, 1.5])
ar = numpy.array([‘a’, ‘b’, ‘c’, ‘d’])
name = numpy.array([‘Rahul’, ‘Sonam’, ‘Raj’], dtype=str)
name = numpy.array([‘Rahul’, ‘Sonam’, ‘Raj’])
```

Creating 1D Array using array ( ) Function

Syntax:-

```
from numpy import *
array_name = array([elements])
```

Ex:-

```
from numpy import *
stu_roll = array([101, 102, 103, 104, 105])
```

index

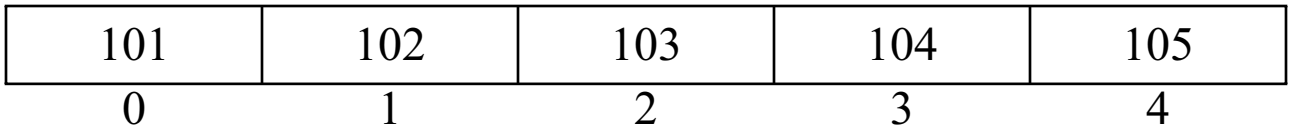
An index represents the position number of an array's element.

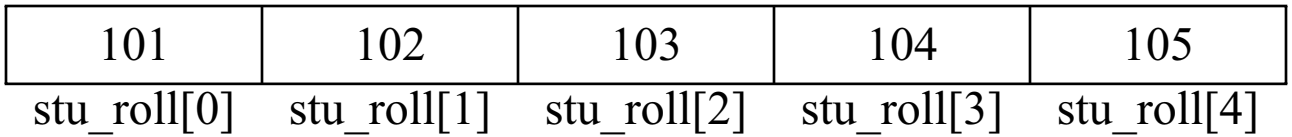
*from numpy import **

stu_roll = array([101, 102, 103, 104, 105])

Python interpreter allocates 5 blocks of memory and stores the elements

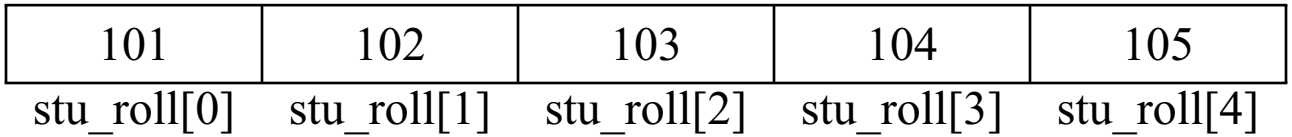
Index always starts with 0





Accessing One-D Array Elements

```
from numpy import *
stu_roll = array([101, 102, 103, 104, 105])
print(stu_roll[0])
print(stu_roll[1])
print(stu_roll[2])
print(stu_roll[3])
print(stu_roll[4])
```



Modifying 1D Array Elements

```
from numpy import *
stu_roll = array([101, 102, 103, 104, 105])
stu_roll[1] = 110
print(stu_roll[1])
```

101	102	103	104	105
stu_roll[0]	stu_roll[1]	stu_roll[2]	stu_roll[3]	stu_roll[4]

Accessing array using for loop

*from numpy import **

stu_roll = array([101, 102, 103, 104, 105])

Without index

```
for element in stu_roll:
        print(element)
```

With index

```
n = len(stu_roll)
for i in range(n):
        print(stu_roll[i])
```

Accessing array using while loop

```
from numpy import *
stu_roll = array([101, 102, 103, 104, 105])

n = len(stu_roll)
i = 0
while i < n :
        print(stu_roll[i])
        i+=1
```

linspace ( ) Function

linspace ( ) Function is used to create an array with evenly spaced numbers between a start point and stop point.

Syntax:-

numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)

Where,

start – It represents starting element.

stop – It represents ending element.

num – It represents number of parts the element should be divided. Default is 50. It must be non-negative.

endpoint- If True, stop is the last element. If False, stop is not included.

https://www.numpy.org/devdocs/reference/generated/numpy.linspace.html

Creating Array using linspace ( ) Function

Syntax:-

from numpy import *

array_name = linspace(start, stop, num=50, endpoint=True)

Ex:-

from numpy import *

a = linspace(1, 8)

a = linspace(1, 8, num=5)

a = linspace(1, 8, 5)

a = linspace(1, 8, 5, endpoint=False)

1.0	2.75	4.5	6.25	8.0
a[0]	a[1]	a[2]	a[3]	a[4]

Accessing One-D Array Elements

from numpy import *

a = linspace(1, 8, 5)

print(*a*[0])

print(*a*[1])

print(*a*[2])

print(*a*[3])

print(*a*[4])

1.0	2.75	4.5	6.25	8.0
a[0]	a[1]	a[2]	a[3]	a[4]

Accessing using for loop

```
from numpy import *
a = linspace(1, 8, 5)
```

Without index

```
for el in a:
        print(el)
```

With index

```
n = len(a)
for i in range(n):
        print(a[i])
```

Accessing using while loop

```
from numpy import *
a = linspace(1, 8, 5)

n = len(a)
i = 0
while i < n :
        print(a[i])
        i+=1
```

logspace ( ) Function

logspace ( ) Function is used to create an array with evenly spaced numbers logarithmically. The sequence starts at base ** start (base to the power of start) and ends with base ** stop.

Syntax:-

numpy.logspace(start, stop, num=50, endpoint=True, base=10.0, dtype=None, axis=0)

Where,

start – It represents starting element which will become base to the power of start $(base^{start})$

stop – It represents ending element which will become base to the power of stop $(base^{stop})$

num – It represents number of parts the element should be divided. Default is 50. It must be non-negative.

endpoint- If True, stop is the last element. If False, stop is not included.

base - The base of the log space.

dtype – The type of output array.

Creating Array using logspace ( ) Function

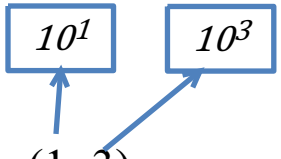
Syntax:-

from numpy import *

array_name = logspace(start, stop, num=50, endpoint=True, base=10.0 dtype=None)

Ex:-

a = logspace(1, 3)



10.0	31.62	100.0	316.2	1000.0
a[0]	a[1]	a[2]	a[3]	a[4]

from numpy import *

a = logspace(1, 3)

a = logspace(1, 3, 5)

a = logspace(1, 3, 5, base=12.0)

Accessing One-D Array Elements

*from numpy import **

a = logspace(1, 3, 5)

print(*a*[0])

print(*a*[1])

print(*a*[2])

print(*a*[3])

print(*a*[4])

10.0	31.62	100.0	316.2	1000.0
a[0]	a[1]	a[2]	a[3]	a[4]

Accessing using for loop

```
from numpy import *
a = logspace(1, 3, 5)
```

Without index

```
for el in a:
        print(el)
```

With index

```
n = len(a)
for i in range(n):
        print(a[i])
```

Accessing using while loop

```
from numpy import *
a = logspace(1, 3, 5)

n = len(a)
i = 0
while i < n :
        print(a[i])
        i+=1
```

arange( ) Function

arange ( ) Function is used to create an array with a group of elements from start to one element prior to stop in steps of stepsize.

Syntax:-

numpy.arange(start, stop, stepsize, dtype=None)

Where,

start – Start of interval. The interval includes this value. The default start value is 0.

stop – End of interval. The interval does not include this value, except in some cases where step is not an integer and floating point round-off affects the length of out.

stepsize – Spacing between values. The default step size is 1.

dtype – The type of output array.

Creating Array using arange ( ) Function

Syntax:-

from numpy import *

array_name = arange(start, stop, stepsize, dtype=None)

Ex:-

from numpy import *

a = arange(5)

a = arange(5.0)

a = arange(1, 6)

a = arange(1, 10, 2,)

0.0	1.0	2.0	3.0	4.0
a[0]	a[1]	a[2]	a[3]	a[4]

Accessing One-D Array Elements

*from numpy import **

a = arange(5)

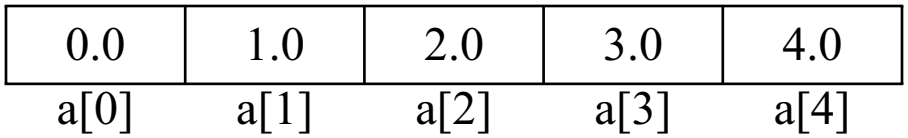
print(*a*[0])

print(*a*[1])

print(*a*[2])

print(*a*[3])

print(*a*[4])



Accessing using for loop

```
from numpy import *
a = arange(5)
```

Without index

```
for el in a:
        print(el)
```

With index

```
n = len(a)
for i in range(n):
        print(a[i])
```

Accessing using while loop

```
from numpy import *
a = arange(5)

n = len(a)
i = 0
while i < n :
        print(a[i])
        i+=1
```

zeros( ) Function

zeros ( ) Function is used to create an array with all zeros.

Syntax:-

numpy.zeros(shape, dtype=float, order='C')

Where,

shape – shape of new array. It can be an int which will represent number of elements or can be tuple of int. ex:- 5, (5,) (3, 1)

dtype – The desired data-type for the array.

Order - Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory. It can be C or F.

Creating Array using zeros ( ) Function

Syntax:-

from numpy import *

array_name = zeros(shape, dtype=float)

Ex:-

from numpy import *

a = zeros(5)

a = zeros(5, dtype=int)

a = zeros((3, 2))

0.0	0.0	0.0	0.0	0.0
a[0]	a[1]	a[2]	a[3]	a[4]

Accessing One-D Array Elements

*from numpy import **

a = zeros(5)

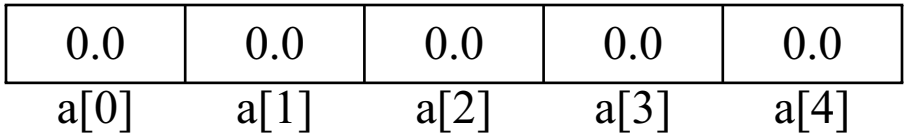
print(*a*[0])

print(*a*[1])

print(*a*[2])

print(*a*[3])

print(*a*[4])



Accessing using for loop

```
from numpy import *
a = zeros(5)
```

Without index

```
for el in a:
        print(el)
```

With index

```
n = len(a)
for i in range(n):
        print(a[i])
```

Accessing using while loop

```
from numpy import *
a = zeros(5)

n = len(a)
i = 0
while i < n :
        print(a[i])
        i+=1
```

ones( ) Function

ones ( ) Function is used to create an array with all 1s.

Syntax:-

numpy.ones(shape, dtype=float, order='C')

Where,

shape – shape of new array. It can be an int which will represent number of elements or can be tuple of int.

dtype – The desired data-type for the array. Default is float.

Order - Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory. It can be C or F.

Creating Array using ones ( ) Function

Syntax:-

from numpy import *

array_name = ones(shape, dtype=float)

Ex:-

from numpy import *

a = ones(5)

a = ones(5, dtype=int)

a = ones((3, 2))

1.	1.	1.	1.	1.
a[0]	a[1]	a[2]	a[3]	a[4]

Accessing One-D Array Elements

*from numpy import **

a = ones(5)

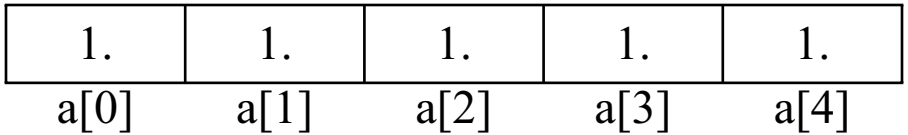
print(*a*[0])

print(*a*[1])

print(*a*[2])

print(*a*[3])

print(*a*[4])



Accessing using for loop

```
from numpy import *
a = ones(5)
```

Without index

```
for el in a:
        print(el)
```

With index

```
n = len(a)
for i in range(n):
        print(a[i])
```

Accessing using while loop

```
from numpy import *
a = ones(5)

n = len(a)
i = 0
while i < n :
        print(a[i])
        i+=1
```

Mathematical Operations on Arrays using numpy

We can perform mathematical operations like addition, subtraction, multiplication, division etc on the elements of an array. Math functions from the math module can also be possible to apply to the elements of the array.

Ex:-

```
a = array([101, 102, 103, 104, 105])
a = a + 5
```

```
a = array([101, 102, 103, 104, 105])
b = array([10, 20, 30, 40, 50])
c = a + b
```

Relational/ Comparison Operators

Relational operators are used to compare the value of operands (expressions) to produce a logical value. A logical value is either True or False.

Operators	Meaning	Example	Result
<	Less than	5<2	False
>	Greater than	5>2	True
<=	Less than or equal to	5<=2	False
>=	Greater than or equal to	5>=2	True
==	Equal to	5==2	False
!=	Not equal to	5!=2	True

Comparing Arrays using numpy

Comparison operators can be used to compare arrays. The size of array must be same. Comparison operators compares the corresponding elements of the arrays and returns another array with Boolean value.

a = array([100, 200, 300, 400, 500])

b = array([100, 20, 30, 400, 50])

c = a == b

- any ( ) Function – This function returns True, if any one element of the iterable is True. If iterable is empty then returns False.

  Ex:-

  ```
  a = array([100, 200, 300, 400, 500])
  b = array([100, 20, 30, 400, 50])
  c = a == b
  any(c)
  ```

- all ( ) Function – This function returns True, if all element of the iterable are True or iterable is empty.

  Ex:-

  ```
  a = array([100, 200, 300, 400, 500])
  b = array([100, 200, 300, 400, 500])
  c = a == b
  all(c)
  ```

Note - These are python's built-in Functions

- where ( ) Function- This function is used to create a new Array which contains, returned element chosen from expression1 or expression2 depending on condition. If condition is True expression1 is executed else expression 2.

  Syntax:- numpy.where(condition, expression1, expression2)

  Ex:-

  a = array([100, 200, 300, 400, 500])

  b = array([10, 201, 30, 40, 50])

  c = where(a>b, a, b)

- nonzero ( ) Function- This function is used to determine the positions of elements which are non zero. This function returns an array that contains the indexes of the element of the array which are not equal to zero

  Syntax:- numpy.nonzero(a)

  Ex:-

  a = array([100, 200, 300, 400, 500])

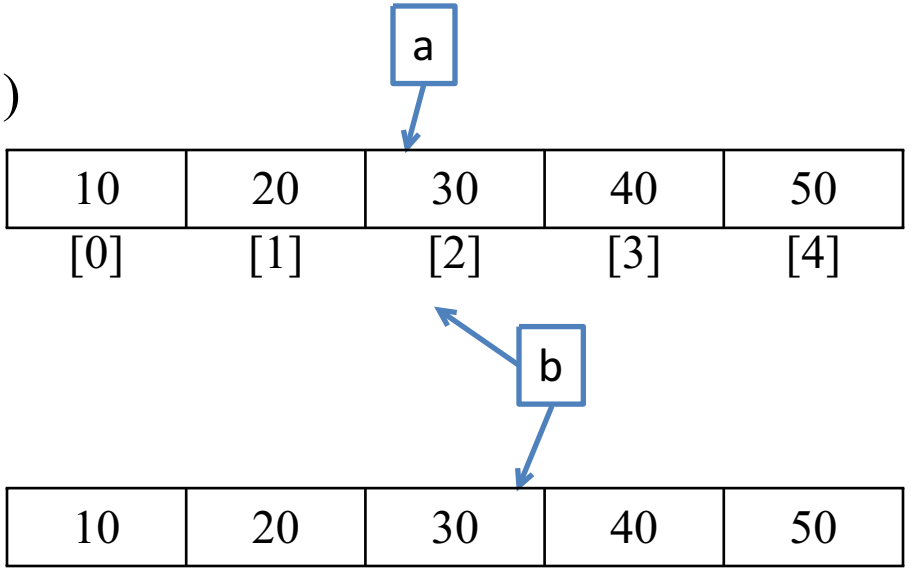
  c = nonzero(a)

Aliasing Array

Aliasing means giving another name to the existing object. It doesn't mean copying.

a = array([10, 20, 30, 40, 50])

b = a



view ( ) Method

view ( ) – This method is used to construct a new view of array with same data of existing array. The existing array and new array will share different memory locations.

If the new array get modified, the existing will also be modified as the elements in both the arrays will be like mirror image.

Ex:-

```
a = array([10, 20, 30, 40, 50])
b = a.view( )
```

copy ( ) Method

copy ( ) – This method is used to create a copy of an existing array. If the new array get modified, the existing array will not be affected or vice versa. Both the arrays are independent.

Ex:-

```
a = array([10, 20, 30, 40, 50])
b = copy(a)
```

Multi-Dimensional Array

The 2D arrays, 3D arrays, etc are called Multi-dimensional Arrays.

Two Dimensional Array

If an array contains more than 1 row and 1 column that is known as Two Dimensional Array. It is also known as array of arrays.

Ex:-

```
a = array([   [10, 20, 30, 40],
              [50, 60, 70, 80]     ])
```

Similarly we can create 3D array.

Ex:-

```
a = array ([  [  [2, 5, 8], [6, 4, 3]   ],
              [  [3, 7, 9], [5, 4, 9]  ]   ])
```

Way of creating Multi-D Array

- array ( ) Function
- zeros ( ) Function
- ones ( ) Function
- reshape ( ) Function

array ( ) Function

array ( ) Function of numpy is used to create a multi dimensional array.

Syntax:-

numpy.array(object, dtype=None, copy=True, order='K', subok=False, ndmin=0)

https://www.numpy.org/devdocs/reference/generated/numpy.array.html

Creating 2D Array using array ( ) Function

Syntax:-


```
import numpy
array_name = numpy.array([  [elements],  [elements]  ])
```

Ex:-

```
import numpy
a = numpy.array([   [10, 20, 30, 40],  [50, 60, 70, 80]     ])
a = numpy.array([   [10, 20, 30, 40],
                    [50, 60, 70, 80]     ])
a = numpy.array([   ['Rahul', 'Sonam', 'Raj'],
                    ['Dell', 'Asus', 'Lenovo']     ], dtype=str)
```

Creating and Initializing 2D Array

Syntax:-

```
from numpy import *
array_name = array([  [elements],  [elements]  ])
```

Ex:-

```
from numpy import *
a = array([   [10, 20, 30, 40],
              [50, 60, 70, 80]   ])
```

index

An index represents the position number of an array's element.

```
from numpy import *
a = array([  [10, 20, 30, 40],
             [50, 60, 70, 80]  ])
```

Python interpreter allocates 8 blocks of memory and stores the elements

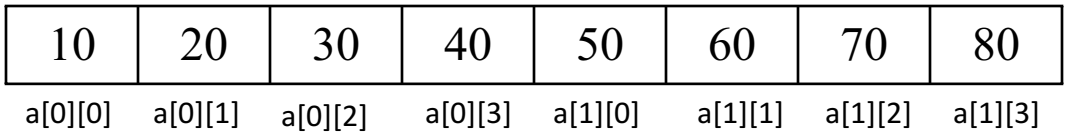
Index always starts with 0

	0	1	2	3
0	10	20	30	40
1	50	60	70	80

10	20	30	40	50	60	70	80
[0][0]	[0][1]	[0][2]	[0][3]	[1][0]	[1][1]	[1][2]	[1][3]

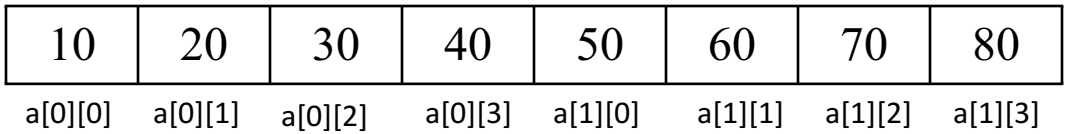
Accessing 2D Array Elements

```
from numpy import *
a = array([   [10, 20, 30, 40],
              [50, 60, 70, 80]   ])
print(a[0][0])
print(a[0] [1])
print(a[0] [2])
print(a[0] [3])
print(a[1] [0])
print(a[1] [1])
print(a[1] [2])
print(a[1] [3])
```



Modifying 2D Array Elements

```
from numpy import *
a = array([   [10, 20, 30, 40],
              [50, 60, 70, 80]   ])
a[1][2] = 100
print(a[1][2])
```



Accessing 2D Array using for loop

```
from numpy import *
a = array([  [10, 20, 30, 40],
             [50, 60, 70, 80]  ])
```

Without index

```
for r in a:
    for c in r:
        print(c)
    print( )
```

Outer for loop

Inner for loop

With index

```
n = len(a)
for i in range(n):
    for j in range(len(a[i])):
        print(a[i][j])
    print ( )
```

Inner for loop

Outer for loop

The outer for loop represents the rows and the inner for loop represents the columns in each row.

Accessing 2D Array using while loop

```
from numpy import *
a = array([   [10, 20, 30, 40],
              [50, 60, 70, 80]   ])
n = len(a)
i = 0
while i < n :
        j = 0
        while j < len(a[i]):
                print(a[i][j])
                j+=1
        i+=1
```

zeros( ) Function

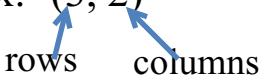
zeros ( ) Function is used to create 2D array with all zeros.

Syntax:-

numpy.zeros(shape, dtype=float, order='C')

Where,

shape – shape of new array. It can be an int which will represent number of elements or can be tuple of int. ex:- (3, 2)



dtype – The desired data-type for the array.

Order - Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory. It can be C or F.

Creating 2D Array using zeros ( ) Function

Syntax:-

from numpy import *

array_name = zeros(shape, dtype=float)

Ex:-

from numpy import *

a = zeros((3, 2))

0.	0.	0.	0.	0.	0.
a[0][0]	a[0][1]	a[1][0]	a[1][1]	a[2][0]	a[2][1]

```
a = array([  [0., 0.],
             [0., 0.],
             [0., 0.]  ])
```

Accessing 2D Array Elements

*from numpy import **

a = zeros((3, 2))

print(*a*[0][0])

print(*a*[0][1])

print(*a*[1][0])

print(*a*[1][1])

print(*a*[2][0])

print(*a*[2][1])

```
a = array([ [0., 0.],
            [0., 0.],
            [0., 0.] ])
```

	[0]	[1]
[0]	0.	0.
[1]	0.	0.
[2]	0.	0.

Accessing 2D Array using for loop

```
from numpy import *
a = zeros((3, 2))
```

```
a = array([ [0., 0.],
            [0., 0.],
            [0., 0.] ])
```

Without index

```
for r in a:
    for c in r:
        print(c)
    print( )
```

Outer for loop

Inner for loop

With index

```
n = len(a)
for i in range(n):
    for j in range(len(a[i])):
        print(a[i][j])
    print ( )
```

Inner for loop

Outer for loop

The outer for loop represents the rows and the inner for loop represents the columns in each row.

Accessing 2D Array using while loop

```
from numpy import *
a = zeros((3, 2))
n = len(a)
i = 0
while i < n :
        j = 0
        while j < len(a[i]):
                print(a[i][j])
                j+=1
        i+=1
```

```
a = array([  [0., 0.],
             [0., 0.],
             [0., 0.]  ])
```

ones( ) Function

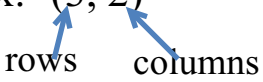
ones ( ) Function is used to create 2D array with several rows and columns with all 1s.

Syntax:-

numpy.ones(shape, dtype=float, order='C')

Where,

shape – shape of new array. It can be an int which will represent number of elements or can be tuple of int. ex:- (3, 2)



dtype – The desired data-type for the array. Default is float.

Order - Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory. It can be C or F.

Creating 2D Array using ones ( ) Function

Syntax:-

from numpy import *

array_name = ones(shape, dtype=float)

Ex:-

from numpy import *

a = ones((3, 2))

1.	1.	1.	1.	1.	1.
a[0][0]	a[0][1]	a[1][0]	a[1][1]	a[2][0]	a[2][1]

```
a = array([  [1., 1.],
             [1., 1.],
             [1., 1.]  ])
```

Accessing 2D Array Elements

```
from numpy import *
a = ones((3, 2))
print(a[0][0])
print(a[0][1])
print(a[1][0])
print(a[1][1])
print(a[2][0])
print(a[2][1])
```

```
a = array([ [1., 1.],
            [1., 1.],
            [1., 1.] ])
```

	[0]	[1]
[0]	1.	1.
[1]	1.	1.
[2]	1.	1.

Accessing 2D Array using for loop

```
from numpy import *
a = ones((3, 2))
```

```
a = array([ [1., 1.],
            [1., 1.],
            [1., 1.] ])
```

Without index

```
for r in a:
    for c in r:
        print(c)
    print( )
```

Outer for loop

Inner for loop

With index

```
n = len(a)
for i in range(n):
    for j in range(len(a[i])):
        print(a[i][j])
    print ( )
```

Inner for loop

Outer for loop

The outer for loop represents the rows and the inner for loop represents the columns in each row.

Accessing 2D Array using while loop

```
from numpy import *
a = ones((3, 2))
n = len(a)
i = 0
while i < n :
        j = 0
        while j < len(a[i]):
                print(a[i][j])
                j+=1
        i+=1
```

```
a = array([  [1., 1.],
             [1., 1.],
             [1., 1.]  ])
```

reshape ( ) Function

This function is used to change the shape of array. We can convert 1D array to 2D or 3D array and vice versa. The new array should have the same number of elements as in the original array.

Syntax:-

reshape(array_name, (n, r, c))

Where,

array_name – It represents the name of the array whose elements to be converted.

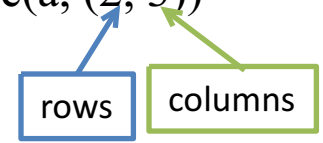
n – n is the number of arrays in the resultant array

r – r is the number of rows

c – c is the number of columns

Convert 1D Array to 2D Array

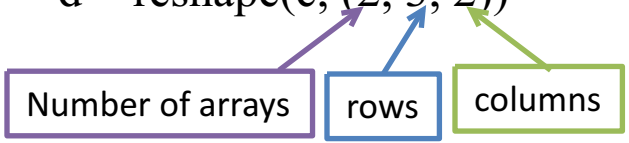
```
a = array([1, 2, 3, 4, 5, 6])
b = reshape(a, (2, 3))
```




```
b = [  [1, 2, 3]
       [4, 5, 6]   ]
```

Convert 1D Array to 3D Array

```
c = array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
d = reshape(c, (2, 3, 2))
```




```
d = [ [ [1, 2],
        [3, 4],
        [5, 6] ],
      [ [7, 8],
        [9, 10],
        [11, 12] ]  ]
```

Convert 2D Array to 1D Array

e = array([[1, 2, 3], [4, 5, 6]])

f = reshape(e, (6))

f = [1, 2, 3, 4, 5, 6]

flatten ( ) Method

The flatten ( ) method is used to convert 2D or 3D array to 1D array.

Syntax:- array_name.flatten()

Ex-

```
e = array([[1, 2, 3], [4, 5, 6]])
f = e.flatten()
```

Attributes of Numpy Array

ndim – This attribute represents the number of dimensions or axes of the array. The number of dimensions is also referred as Rank.

Syntax:- array_name.ndim

shape – This attribute represents the shape of an array. The shape is a tuple listing the number of elements along each dimension.

Syntax:- array_name.shape

Attributes of Numpy Array

size – This attribute represents the total number of elements in the array.

Syntax:- array_name.size

itemsize – This attribute represents the memory size of the array element in bytes.

Syntax:- array_name.itemsize

Attributes of Numpy Array

dtype – This attribute represents the datatype of elements in the array.

Syntax:- array_name.dtype

nbytes – This attribute represents the total number of bytes occupied by an array.

Total number of bytes = size of array * item size of each element in the array

Syntax:- array_name.nbytes

Slicing on 2D Array

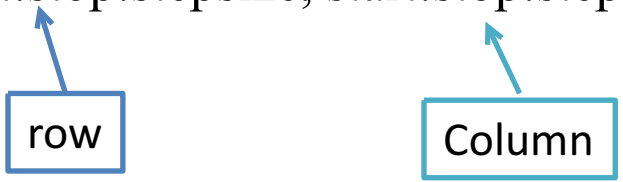
Slicing on arrays can be used to retrieve a piece of the array that contains a group of elements. Slicing is useful to retrieve a range of elements.

Syntax:-

new_array_name = array_name[start:stop:stepsize, start:stop:stepsize]

The default start value is 0

The default stepsize is 1



Numpy Built in Math Functions

- sum(arr) – Returns sum of all the elements in the array arr.
- prod(arr) – Returns product of all the elements in the array arr.
- sqrt(arr) – Returns square root value of each element in the array arr.

https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.math.html

String

String – String represents group of characters. Strings are enclosed in double quotes or single quotes. The *str* data type represents a String.

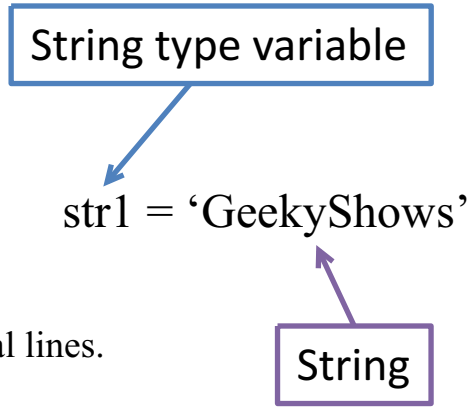
Ex:- “Hello”, “GeekyShows”, ‘Rahul’

str1 = “GeekyShows”

Creating String

Single Quotes
str1 = ‘GeekyShows’

Double quotes
str2 = “GeekyShows”



Triple Single Quotes – This is useful to create strings which span into several lines.
str3 = ‘‘‘ Hello Guys
Please Subscribe
Geekyshows’’’

Triple Double Quotes – This is useful to create strings which span into several lines.
str4 = “““Hello Guys
Please Subscribe
Geekyshows ”””

Creating String

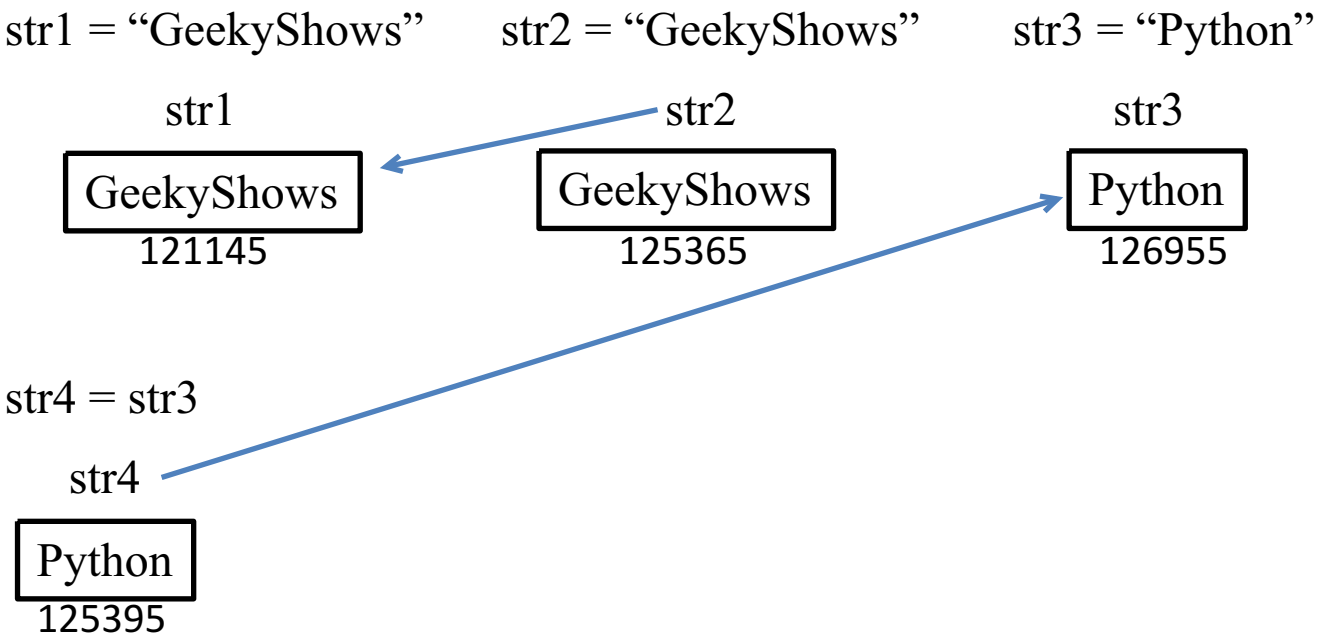
Double Quote inside Single Quotes
str5 = ‘Hello “Geeky Shows” How are you’

Single Quote inside Double quotes
str6 = “Hello ‘Geeky Shows’ How are you”

Using Escape Characters
str7 = “Hello \nHow are You ?”

Raw String – Row string is used to nullify the effect of escape characters.
str8 = r“Hello \nHow are You ?”

Memory Allocation



Memory Allocation

str1 = “GeekyShows”

str1

GeekyShows

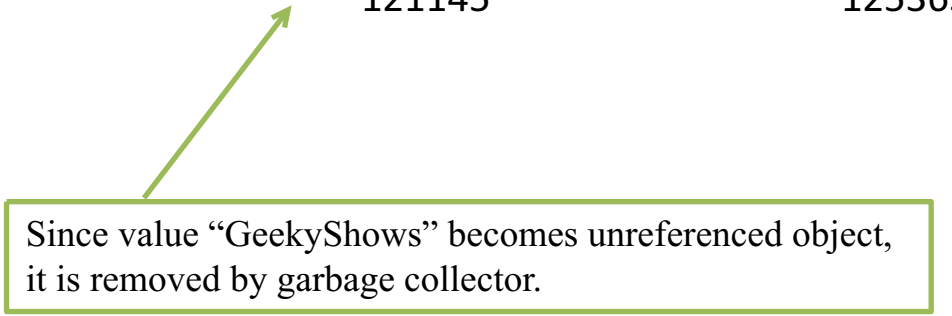
121145

str1 = “Python”

str1

Python

125365

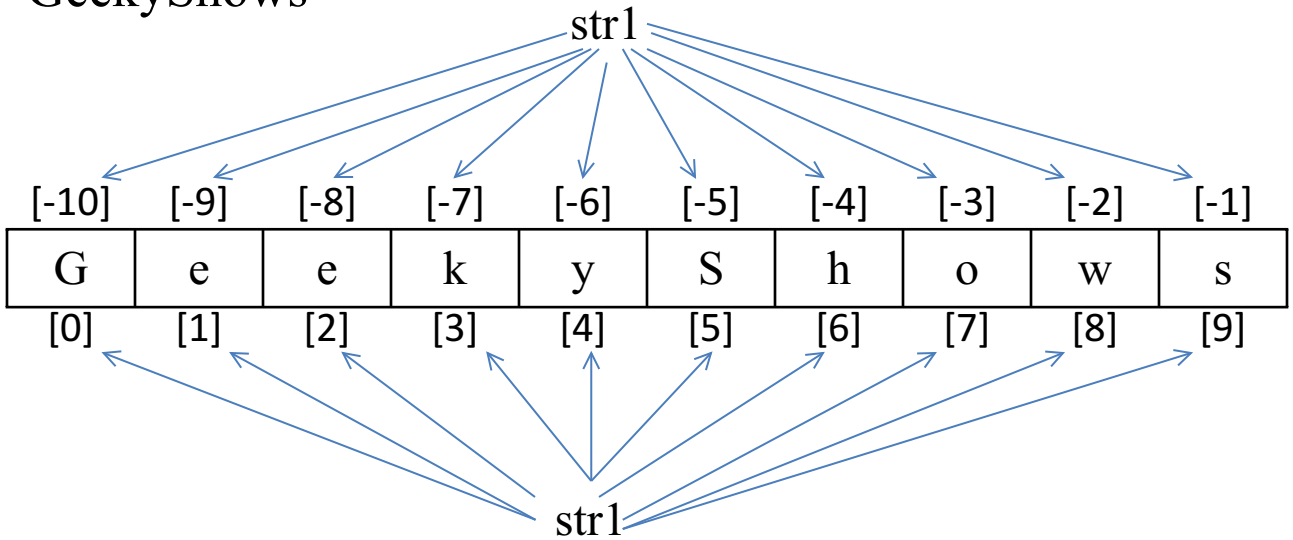


Index

Index represents the position number of characters in a string.

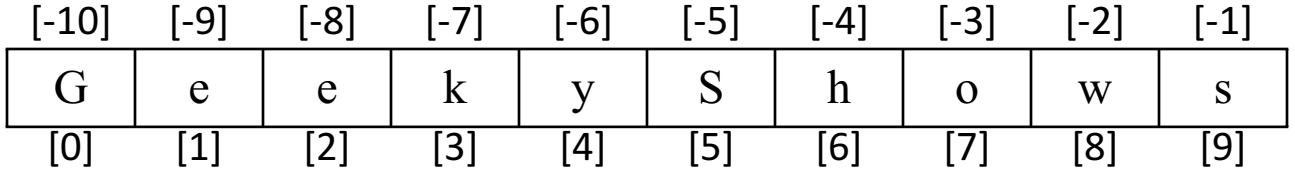
Ex:-

str1 = “GeekyShows”



Accessing Character and String

str1 = “GeekyShows”




```
print(str1[0])
print(str1[1])
print(str1[2])
print(str1[3])
print(str1[-1])


print(str1)
```

String Length

Length of string represents the number of characters in a string.

len( ) Function is used to get length of string.

Ex:-

str1 = “GeekyShows”

n = len(str1)

Access using Loop

```
str1 = “GeekyShows”
for c in str1:
        print(c)

for i range(len(str1)):
        print(str1[i])

i = 0
while i < len(str1) :
        print(str1[i])
        i+=1
```

Mutable and Immutable Object

- Mutable Object – Mutable objects are those object whose value or content can be changed as and when required.

  Ex:- List, Set, Dictionaries

- Immutable Object – Immutable objects are those object whose value or content can not be changed.

  Ex:- Numbers, String, Tuple

Immutable String

In Python, Strings are immutable object which means it's value or content can not be changed.

str1 = "GeekyShows"

[-10]	[-9]	[-8]	[-7]	[-6]	[-5]	[-4]	[-3]	[-2]	[-1]
G	e	e	k	y	S	h	o	w	s
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

str1[4] ="i"

TypeError

Slicing String

Slicing on String can be used to retrieve a piece of the string. Slicing is useful to retrieve a range of elements.

Syntax:-

new_string_name = string_name[start:stop:stepsize]

- start – It represents starting point. By default its 0
- stop – It represents ending point.
- stepsize – It represents step interval. By default It is 1
- If start and stop are not specified then slicing is done from 0^{th} to $n-1^{th}$ elements.
- Start and Stop can be negative number.

Repetition Operator

Repetition operator is used to repeat the string for several times. It is denoted by *

Ex:-

“$” * 7

str1 = “Geeky Shows ”

str1 * 5

Concatenation Operator

Concatenation operator is used to join two string. It is denoted by +

Ex:-

“Geeky” + “Shows”

str1 = “Geeky ”

str2 = “Shows”

str1 + str2

Comparing String

str1 = “GeekyShows”
str2 = “GeekyShows”
result = str1 == str2

str1 = “GeekyShows”
str2 = “Python”
result = str1 == str2

str1 = “A”
str2 = “B”
result = str1 < str2

Operators	Meaning
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Formatting String

- C-Style String Formatting
- format ( ) Method
- f-String / Formatted String Literals

C-Style String Formatting

% operator/ Modulo Operator/ Interpolation Operator – This operator is used for formatting strings. It interprets the left argument much like a sprintf( ) style format string to be applied to the right argument, and returns the string resulting from this formatting operation.

Syntax:- print(“format placeholder” %(data))

Format placeholder = %[flags][width][.precision]type

% - marks the start of the specifier

Flags – It affect the result of some conversion type

Width – Minimum field width

Precision – Given as . Followed by the precision

Type – Conversion type

C-Style String Formatting

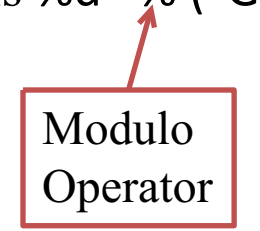
Syntax:- print("format placeholder" %(data))

Format placeholder = %[flags][width][.precision]type

data – It can be literal, variable, expression etc.

Ex:-

print("My name is %s and My age is %d" % ("GeekyShows", 62))



Maintain Order in above statement first string then integer

C-Style String Formatting

Syntax:- print(“format placeholder” % {‘key’:value})

Format placeholder = %(mapping_key)[flags][width][.precision]type

% - marks the start of the specifier

Mapping_key – Consisting of parenthesized sequence of characters

Flags – It affect the result of some conversion type

Width – Minimum field width

Precision – Given as . Followed by the precision

Type – Conversion type

C-Style String Formatting

Example:-

print(“My name is %(nm)s and My age is %(ag)d” % {‘nm’:“GeekyShows”, ‘ag’:62})

print(“My name is %(nm)s and My age is %(ag)d” % {‘ag’:62, ‘nm’:“GeekyShows”})

Do not need to maintain Order in above statement.

C-Style String Formatting

Conversion Type	Meaning
d	Signed integer decimal.
I	Signed integer Decimal
o	Signed octal value.
x	Signed hexadecimal (lowercase).
X	Signed hexadecimal (uppercase).
e	Floating point exponential format (lowercase).
E	Floating point exponential format (uppercase).
f	Floating point decimal format.
F	Floating point decimal format.

C-Style String Formatting

Conversion Type	Meaning
g	Floating Point format. Uses lower case exponential format if exponent is less than -4 or not less than precision, decimal format otherwise
G	Floating Point format. Uses lower case exponential format if exponent is less than -4 or not less than precision, decimal format otherwise
c	Single character (accepts integer or single character string)
r	String (converts any python object using repr())
s	String (converts any Python object using str()).
A	String (Converts any python object using ascii())
%	No argument is converted, result in a % character in the result

C-Style String Formatting

Flag	Meaning
#	Used with o, x or X specifiers the value is preceded with 0, 0o, 0O, 0x or 0X respectively.
0	The conversion will be zero padded for numeric values.
-	The converted value is left adjusted (overrides the '0' conversion if both are given).
' '	(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.
+	A sign character ('+' or '-') will precede the conversion (overrides a “space” flag).

C-Style String Formatting

print(“%d” % 432)	432
print(“%d %d” % (432, 345))	432 345
print(“%f” % 432.123)	432.123000
print(“%f” % 432.123456)	432.123456
print(“%f” % 432.12345651)	432.123457
print(“%f” % 432.12345641)	432.123456
print("%s" % "GeekyShows")	GeekyShows
print("%s %s" % ("Hello", "GeekyShows"))	Hello GeekyShows
print("%d %s" % (432, "GeekyShows"))	432 GeekyShows
#print("%s %d" % (432, "GeekyShows"))	TypeError
print(“%(nm)s %(ag)d” % {‘ag’:432, ‘nm’:“GeekyShows”})	GeekyShows 432

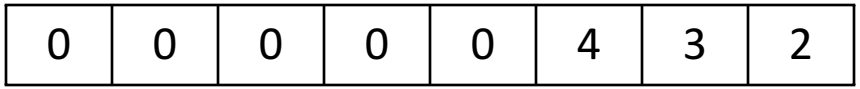
C-Style String Formatting

print("% d" % 432) 432

print(“%+d” % 432) +432

print(“%8d” % 4 3 2)

print(“%08d” % 432) 00000432



C-Style String Formatting

print(“%f” % 432.123) 432.123000

print(“%.3f” % 432.123)

4	3	2	.	1	2	3

print(“%.2f” % 432.123)

4	3	2	.	1	2

print(“%.2f” % 432.128)

4	3	2	.	1	3

C-Style String Formatting

print(“%9.2f” % 432.128)

print(“%09.2f” % 432.123)

print("%9.2f" %4388453232.124)

format ( ) Method

This method is used to format strings. The string on which this method is called can contain literal text or replacement fields delimited by braces {}. Each replacement field contains either the numeric index of a positional argument, or the name of a keyword argument. It returns a copy of the string where each replacement field is replaced with the string value of the corresponding argument.

Syntax:- str.format(arg)

Ex:-

str = “My age is {}”

print(str.format(62))

Ex:- print(“My age is {}”.format(62))

In most of the cases the syntax is similar to the old %-formatting, with the addition of the {} and with : used instead of %.

Ex:-

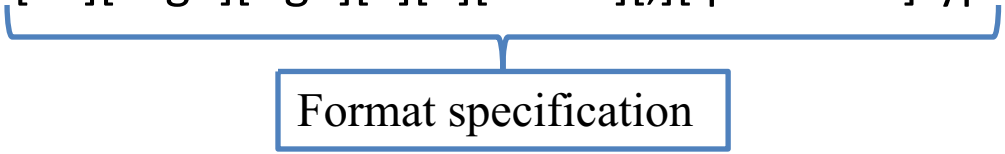
‘%d’ can be translated to ‘{:d}’

'%05.3f' can be translated to '{:05.3f}'.

format ( ) Method

print("Replacement Field".format(values))

RF = {index/key:[fill][align][sign][#][0][width][,][.precision]type}



Ex:-

print("{}".format(10))

print("{} {}".format(10, 20))

print("Mobile Price{} Computer Price {}".format(10, 20))

Format Method

Conversion Type	Meaning
d	Signed integer decimal.
o	Signed octal value.
x	Signed hexadecimal (lowercase).
X	Signed hexadecimal (uppercase).
b	Binary Format
e	Floating point exponential format (lowercase).
E	Floating point exponential format (uppercase).
f	Floating point decimal format. (Default: 6)
F	Same as 'f'. Except displays 'inf' as 'INF' and 'nan' as 'NAN'

Format Method

Conversion Type	Meaning
c	Character. Converts the integer to the corresponding Unicode character before printing
g	General format. Rounds number to p significant digits. (Default precision: 6)
G	Same as 'g'. Except switches to 'E' if the number is large.
n	Same as 'd'. Except it uses current locale setting for number separator
s	String (converts any Python object using str()).
%	Percentage. Multiplies the number by 100 and displays in fixed ('f') format, followed by a percent sign.

Format Method

Alignment Type	Meaning
<	Forces the field to be left-aligned within the available space (This is default for most objects)
^	Forces the field to be centered within the available space.
>	Forces the field to be right-aligned within the available space (This is default for Numbers)
=	Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form ‘+000000120’. This alignment option is only valid for numeric types. It becomes the default when ‘0’ immediately precedes the field width.

Format Method

Sign	Meaning
+	indicates that a sign should be used for both positive as well as negative numbers.
-	indicates that a sign should be used only for negative numbers (this is the default behavior).
	(a space) indicates that a leading space should be used on positive numbers, and a minus sign on negative numbers.

Format Method

- The '**#**' option causes the “alternate form” to be used for the conversion. The alternate form is defined differently for different types. This option is only valid for integer, float, complex and Decimal types. For integers, when binary, octal, or hexadecimal output is used, this option adds the prefix respective '0b', '0o', or '0x' to the output value.

- The '**,**' option signals the use of a comma for a thousands separator. For a locale aware separator, use the 'n' integer presentation type instead.

- The '**_**' option signals the use of an underscore for a thousands separator for floating point presentation types and for integer presentation type 'd'.

Format Method

print(“{}”.format(10))	10
print(“{} {}”.format(10, 20))	10 20
print(“{0}”.format(10))	10
print(“{0} {1}”.format(10, 20))	10 20
print(“{1} {0}”.format(10, 20))	20 10
print(“{num1}”.format(num1=10))	10
print(“{num1} {num2}”.format(num1=10, num2=20))	10 20
print(“{num2} {num1}”.format(num1=10, num2=20))	20 10

Format Method

```
print("{}".format(10.56))
print("{} {}".format(10.56, 20.42))
print("{0}".format(10.56))
print("{0} {1}".format(10.56, 20.42))
print("{1} {0}".format(10.56, 20.42))
print("{num1}".format(num1=10.56))
print("{num1} {num2}".format(num1=10.56, num2=20.42))
print("{num2} {num1}".format(num1=10.56, num2=20.42))
```

Format Method

print(“{}”.format(“GeekyShows”))	GeekyShows
print(“{} {}”.format(“Geeky”, “Shows”))	Geeky Shows
print(“{0}”.format(“Geeky”))	Geeky
print(“{0} {1}”.format(“Geeky”, “Shows”))	Geeky Shows
print(“{1} {0}”.format(“Geeky”, “Shows”))	Shows Geeky
print(“{str1}”.format(str1=“GeekyShows”))	GeekyShows
print(“{str1} {str2}”.format(str1=“Geeky”, str2=“Shows”))	Geeky Shows
print(“{str2} {str1}”.format(str1=“Geeky”, str2=“Shows”))	Shows Geeky

Format Method

```
print(“Hello My Name is {}”.format(“GeekyShows”))
print(“{} {}”.format(10, “Shows”))
print(“{0} {1}”.format(10, “Shows”))
print(“{1} {0}”.format(10, “Shows”))
print(“{num1} {str1}”.format(num1=10, str1=“Shows”))
print(“{str1} {num1}”.format(num1=10, str1=“Shows”))
```

Format Method

```
print(“{}”.format(15))
print(“{:d}”.format(15))
print(“{0:d}”.format(15))
print(“{num1:d}”.format(num1=15))
```

Format Method

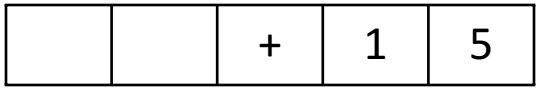
:[fill][align][sign][#][0][width][,][.precision]type

print("{num:5d}".format(num=15))

print("{num:05d}".format(num=15))

0	0	0	1	5

print("{num:+5d}".format(num=15))



print("{num:<5d}".format(num=15))

1	5			

print("{num:*<5d}".format(num=15))

1	5	*	*	*

print("{num:*>5d}".format(num=15))

*	*	*	1	5

print("{num:^5d}".format(num=15))

	1	5		

Format Method

:[fill][align][sign][#][0][width][,][.precision]type

print("{num:8f}".format(num=15.65))

1	5	.	6	5	0	0	0	0

print("{num:8.3f}".format(num=15.65))

		1	5	.	6	5	0

print("{num:+8.2f}".format(num=15.65))

		+	1	5	.	6	5

print("{num:<8.2f}".format(num=15.65))

1	5	.	6	5			

print("{num:*<8.2f}".format(num=15.65))

1	5	.	6	5	*	*	*

print("{num:*>8.2f}".format(num=15.65))

*	*	*	1	5	.	6	5

print("{num:^8.2f}".format(num=15.65))

	1	5	.	6	5		

Format Method

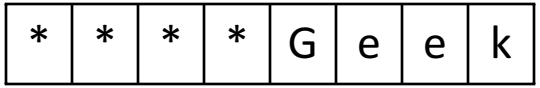
:[fill][align][sign][#][0][width][,][.precision]type

print(“{:8s}”.format(“Geek”))

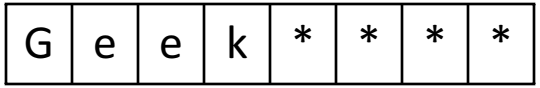
print(“{:>8}”.format(“Geek”))

print(“{:<8}”.format(“Geek”))

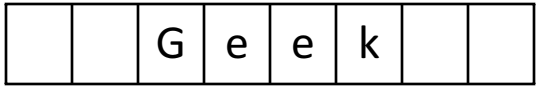
print(“{:*>8s}”.format(“Geek”))



print(“{:*<8}”.format(“Geek”))



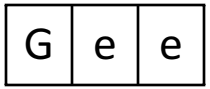
print(“{:^8s}”.format(“Geek”))



Format Method

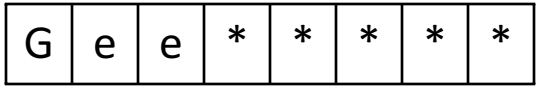
:[fill][align][sign][#][0][width][,][.precision]type

print(“{:.3s}”.format(“GeekShows”))



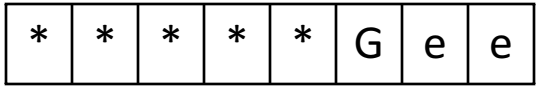
print(“{:8.3}”.format(“GeekShows”))

print(“{:*<8.3}”.format(“GeekyShows”))

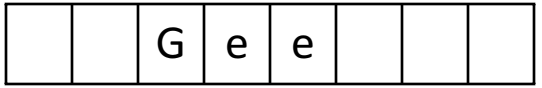


print(“{:>8.3}”.format(“GeekShows”))

print(“{:*>8.3s}”.format(“GeekShows”))



print(“{:^8.3s}”.format(“GeekShows”))



f-String / Formatted String Literal

A formatted string literal or f-string is a string literal that is prefixed with *f* or *F*.

These strings may contain replacement fields, which are expressions delimited by curly braces {}. While other string literals always have a constant value, formatted strings are really expressions evaluated at run time.

Syntax:- f“{index/key/name:[fill][align][sign][#][0][width][,][.precision]type}”

Format specification

Ex:-

a = 10

print(f“{a}”)

Ex:- print(f“My age is {a}”)

f-String / Formatted String Literal

Conversion Type	Meaning
d	Signed integer decimal.
o	Signed octal value.
x	Signed hexadecimal (lowercase).
X	Signed hexadecimal (uppercase).
b	Binary Format
e	Floating point exponential format (lowercase).
E	Floating point exponential format (uppercase).
f	Floating point decimal format. (Default: 6)
F	Same as 'f'. Except displays 'inf' as 'INF' and 'nan' as 'NAN'

f-String / Formatted String Literal

Conversion Type	Meaning
c	Character. Converts the integer to the corresponding Unicode character before printing
g	General format. Rounds number to p significant digits. (Default precision: 6)
G	Same as 'g'. Except switches to 'E' if the number is large.
n	Same as 'd'. Except it uses current locale setting for number separator
s	String (converts any Python object using str()).
%	Percentage. Multiplies the number by 100 and displays in fixed ('f') format, followed by a percent sign.

f-String / Formatted String Literal

Alignment Type	Meaning
<	Forces the field to be left-aligned within the available space (This is default for most objects)
^	Forces the field to be centered within the available space.
>	Forces the field to be right-aligned within the available space (This is default for Numbers)
=	Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form '+000000120'. This alignment option is only valid for numeric types. It becomes the default when '0' immediately precedes the field width.

f-String / Formatted String Literal

Sign	Meaning
+	indicates that a sign should be used for both positive as well as negative numbers.
-	indicates that a sign should be used only for negative numbers (this is the default behavior).
	(a space) indicates that a leading space should be used on positive numbers, and a minus sign on negative numbers.

f-String / Formatted String Literal

- The '**#**' option causes the “alternate form” to be used for the conversion. The alternate form is defined differently for different types. This option is only valid for integer, float, complex and Decimal types. For integers, when binary, octal, or hexadecimal output is used, this option adds the prefix respective '0b', '0o', or '0x' to the output value.

- The '**,**' option signals the use of a comma for a thousands separator. For a locale aware separator, use the 'n' integer presentation type instead.

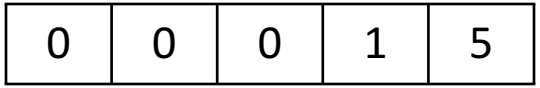
- The '**_**' option signals the use of an underscore for a thousands separator for floating point presentation types and for integer presentation type 'd'.

f-String / Formatted String Literal

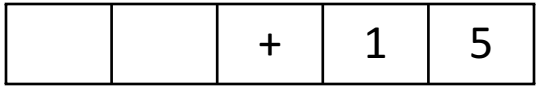
:[fill][align][sign][#][0][width][,][.precision]type

print(f“{num:5d}”)

print(f“{num:05d}”)

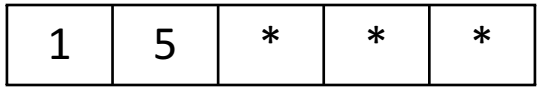


print(f“{num:+5d}”)

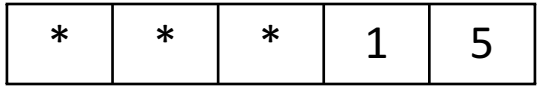


print(f“{num:<5d}”)

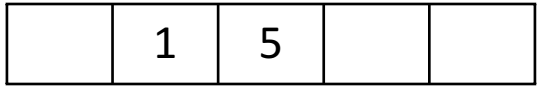
print(f“{num:*<5d}”)



print(f“{num:*>5d}”)



print(f“{num:^5d}”)



f-String / Formatted String Literal

:[fill][align][sign][#][0][width][,][.precision]type

print(f“{num:8f}”)

1	5	.	6	5	0	0	0	0

print(f“{num:8.3f}”)

		1	5	.	6	5	0

print(f“{num:+8.2f}”)

		+	1	5	.	6	5

print(f“{num:<8.2f}”)

1	5	.	6	5			

print(f“{num:*<8.2f}”)

1	5	.	6	5	*	*	*

print(f“{num:*>8.2f}”)

*	*	*	1	5	.	6	5

print(f“{num:^8.2f}”)

	1	5	.	6	5		

f-String / Formatted String Literal

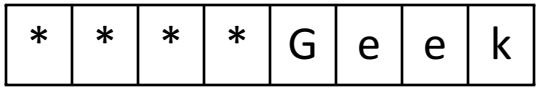
:[fill][align][sign][#][0][width][,][.precision]type
name = “Geek”

print(f“{name:8s}”)

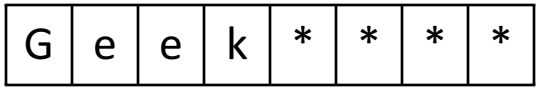
print(f“{name:>8}”)

print(f“{name:<8}”)

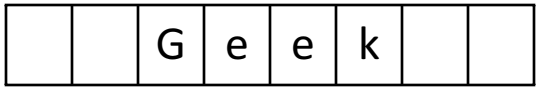
print(f“{name:*>8s}”)



print(f“{name:*<8}”)



print(f“{name:^8s}”)



f-String / Formatted String Literal

:[fill][align][sign][#][0][width][,][.precision]type
name = “GeekyShows”

print(f“{name:.3s}”))

G	e	e

print(f“{name:8.3}”)

G	e	e					

print(f“{name:*<8.3}”)

G	e	e	*	*	*	*	*

print(f“{name:>8.3}”)

					G	e	e

print(f“{name:*>8.3s}”)

*	*	*	*	*	G	e	e

print(f“{name:^8.3s}”)

		G	e	e			

String Functions

- upper ( ) – This method is used to convert all character of a string into uppercase.

  Syntax:- string.upper( )

- lower ( ) – This method is used to convert all character of a string into lowercase.

  Syntax:- string.lower( )

- swapcase ( ) – This method is used to convert all lower case character into uppercase and vice versa.

  Syntax:- string.swapcase ( )

- title ( ) – This method is used to convert the string in such that each word in string will start with a capital letter and remaining will be small letter.

  Syntax:- string.title ( )

String Functions

- isupper ( ) – This method is used to test whether given string is in upper case or not, it returns True if string contains at least one letter and all characters are in upper case else returns False.

  Syntax:- string.isupper( )

- islower ( ) – This method is used to test whether given string is in lower case or not, it returns True if string contains at least one letter and all characters are in lower case else returns False.

  Syntax:- string.islower( )

- istitle ( ) – This method is used to test whether given string is in title format or not, it returns True if string contains at least one letter and each word of the string starts with a capital letter else returns False.

  Syntax:- string.istitle( )

String Functions

- isdigit ( ) – This method returns True if the string contains only numeric digits (0 to 9) else returns False.

  Syntax:- string.isdigit()

- isalpha ( ) – This method returns True if the string has at least one character and all are alphabets (A to Z and a to z) else returns False

  Syntax:- string.isalpha()

- isalnum ( ) – This method returns True if the string has at least one character and all characters in the string are alphanumeric (A to Z, a to z and 0 to 9) else returns False

  Syntax:- string.isalnum()

String Functions

- isspace ( ) – This method returns True if the string contains only space else returns False.

  Syntax:- string.isspace()

String Functions

- lstrip ( ) – This method is used to remove the space which are at left side of the string.

  Syntax:- string.lstrip()

- rstrip ( ) – This method is used to remove the space which are at right side of the string.

  Syntax:- string.rstrip()

- strip ( ) – This method is used to remove the space from the both side of the string.

  Syntax:- string.strip()

String Functions

- replace ( ) – This method is used to replace a sub string in a string with another sub string.

  Syntax:- string.replace(old, new)

- split ( ) – This method is used to split/break a string into pieces. These pieces returns as a list.

  Syntax:- string.split(‘separator’)

- join ( ) – This method is used to join strings into one string.

  Syntax:- “separator”.join(string_list)

String Functions

- startswith ( ) – This method is used to check whether a string is starting with a substring or not. It returns True if the string starts with specified sub string.

  Syntax:- string.startswith(‘specified_string’)

- endswith ( ) – This method is used to check whether a string is ending with a substring or not. It returns True if the string ends with specified sub string.

  Syntax:- string.startswith(‘specified_string’)

Function

Function are subprograms which are used to compute a value or perform a task.

Type of Functions:-

- Built-in Function

  Ex: - print( ), upper( ) etc

- User-defined Function

Advantage of Function

- Write once and use it as many time as you need. This provides code reusability.
- Function facilitates ease of code maintenance.
- Divide Large task into many small task so it will help you to debug code
- You can remove or add new feature to a function anytime.

Function Definition

We can define a function using def keyword followed by function name with parentheses.

This is also called as Creating a Function, Writing a Function, Defining a Function.

Syntax : -

```
def Function_name ( ):
        Local Variable
        block of statement
        return (variable or expression)
```

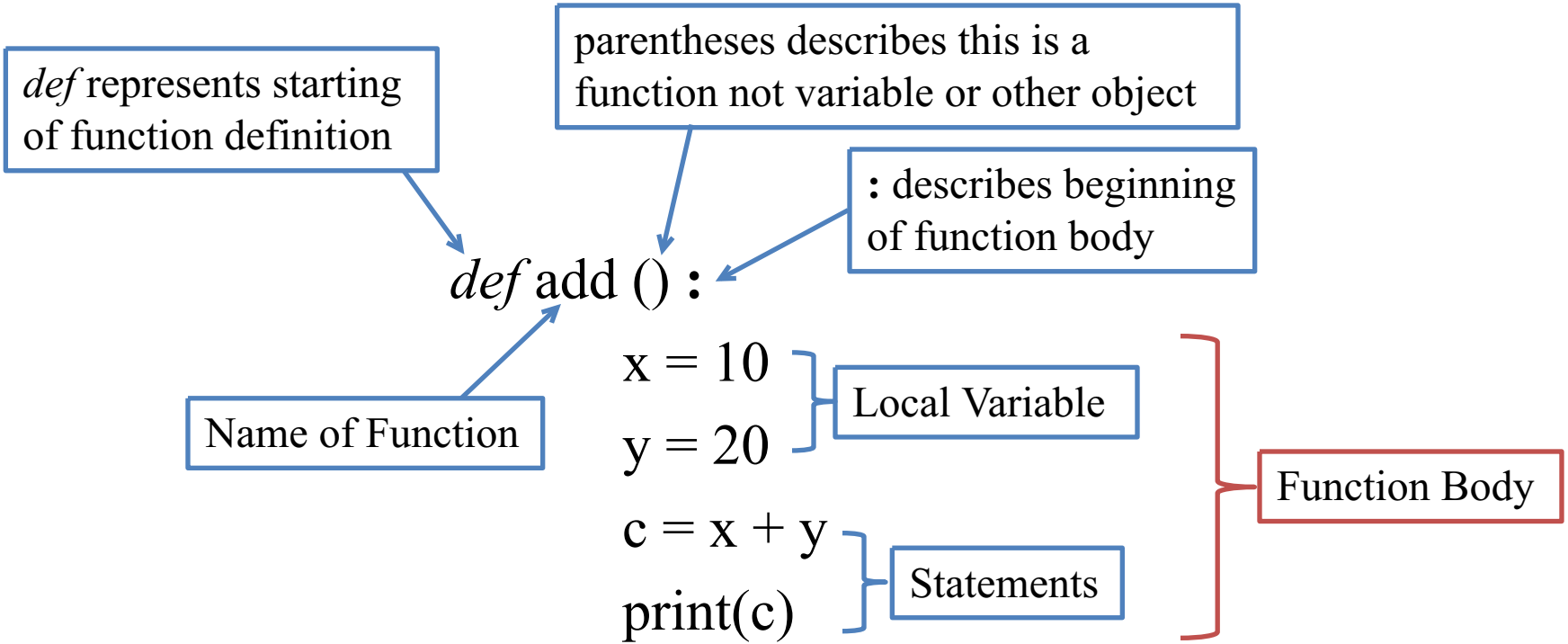
Function Body

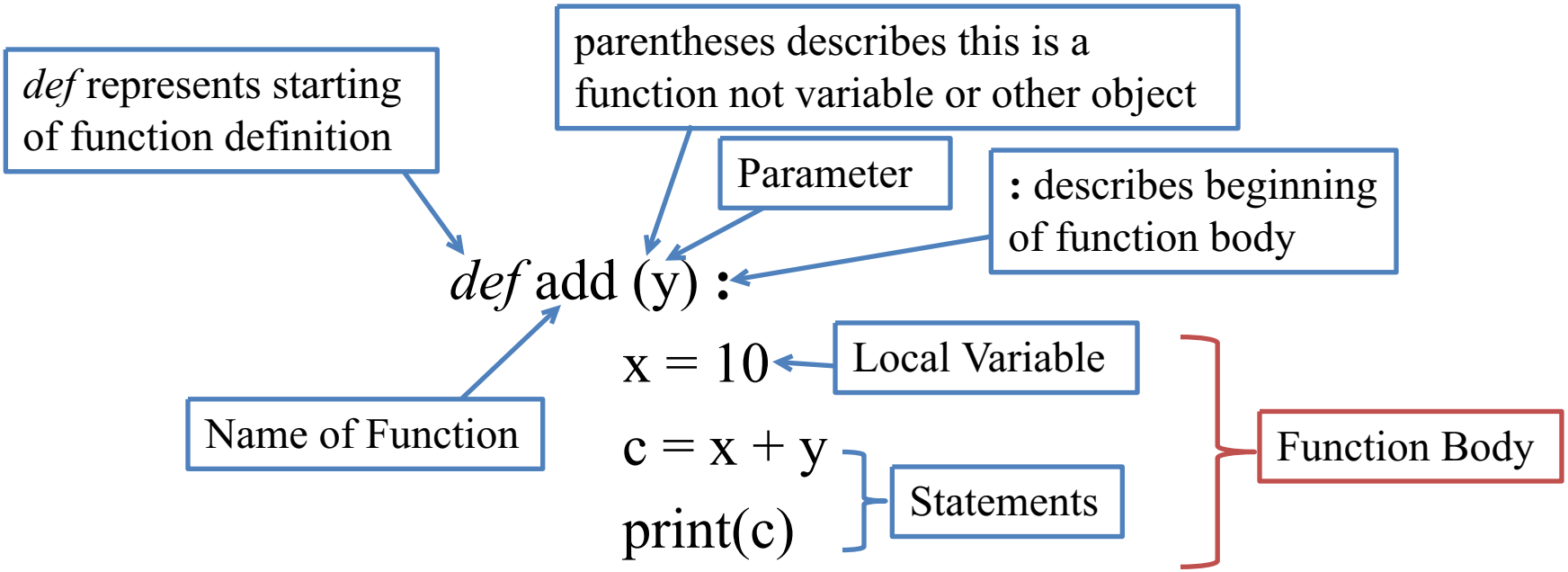
Syntax : -

```
def Function_name (para1, para2, …):
        Local Variable
        block of statement
        return (variable or expression)
```

Function Body

Note – Need to maintain proper indentation


def represents starting of function definition
parentheses describes this is a function not variable or other object
: describes beginning of function body
def add () :
Name of Function
x = 10
y = 20
Local Variable
c = x + y
print(c)
Statements
Function Body


def represents starting of function definition
parentheses describes this is a function not variable or other object
Parameter
: describes beginning of function body
def add (y) :
Name of Function
x = 10
Local Variable
c = x + y
print(c)
Statements
Function Body

Calling a function

A Function runs only when we call it, function can not run on its own.

Syntax:-

function_name ( )

function_name (arg1, arg2, …)

Ex: -

add ( )

add (20)

add(10.56)

item(“GeekyShows”)

```
a = 10
add(a)
```

How Function Work

```
def add () :
        x = 10
        y = 20
        c = x + y
        print(c)
add()
```

```
def add (y) :
        x = 10
        c = x + y
        print(c)
add(20)
```

The parameter *y* do not know which type of value they are about to receive till the value is passed at the time of calling the function. It means the type of data is determined only during runtime not at compile time this is called Dynamic Typing.

Return Statement

Return statements can be used to return something from the function. In Python, it is possible to return one or more variables/values.

Syntax : -

return (variable or expression);

Ex: -

return 50

return (50)

return (x + y)

return (y)

return (2, 4)

return (x, y)

```
def add (y) :
    x = 10
    c = x + y
    return c
sum = add (20)
print(sum)
```

```
def add (y) :
    x = 10
    return x + y
sum = add (20)
print(sum)
```

```
def add_sub (y) :
    x = 10
    c = x + y
    d = y - x
    return c, d
sum, sub = add (20)
print(sum)
print(sub)
```

Nested Function

When we define one function inside another function, it is known as Nested Function or Function Nesting.

Ex:-

```
def disp():
    def show():
        print("Show Function")
    print("Disp Function")
    show()

disp( )
```

Pass a Function as Parameter

We can pass a function as parameter to another function.

Ex:-

```
def disp(sh):
        print("Disp Function" + sh())

def show():
        return " Show Function"

disp(show)
```

Function return another Function

A function can return another function.

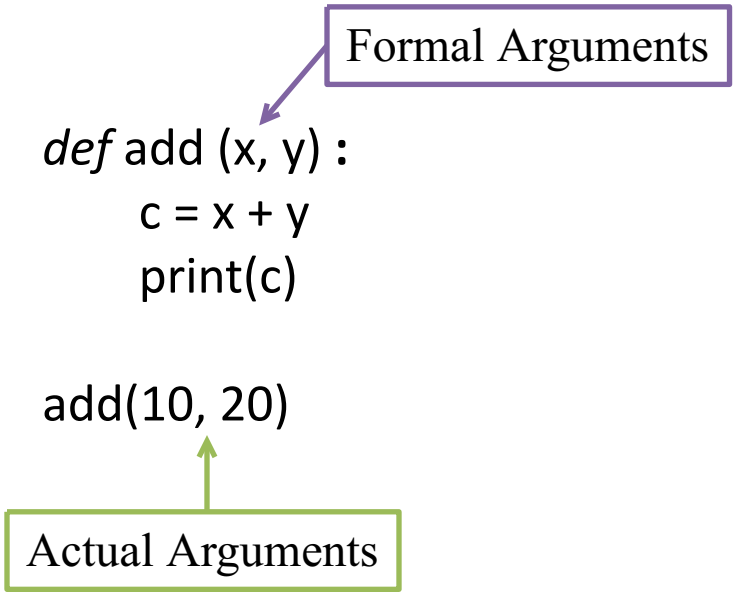
Ex:-

```
def disp():
    def show():
        return "Show Function"
    print("Disp Function")
    return show

r_sh = disp()
print(r_sh())
```

```
def disp(sh):
        print("Disp Function")
        return sh

def show():
        return "Show Function"

r_sh = disp(show)
print(r_sh())
```

Actual and Formal Argument

- Formal Argument - Function definition parameters are called as formal arguments
- Actual Argument - Function call arguments are actual arguments



Type of Actual Arguments

- Positional Arguments
- Keyword Arguments
- Default Arguments
- Variable Length Arguments
- Keyword Variable Length Arguments

Positional Arguments

These arguments are passed to the function in correct positional order.

The number of arguments and their positions in the function definition should be equal to the number and position of the argument in the function call.

```
def pw (x, y) :
    z = x**y
    print(z)

pw(5, 2)
```

```
def pw (x, y) :
    z = x**y
    print(z)

pw(2, 5)
```

```
def pw (x, y) :
    z = x**y
    print(z)

pw(5, 2, 3)
```

Keyword Arguments

These arguments are passed to the function with name-value pair so keyword arguments can identify the formal argument by their names.

The keyword argument's name and formal argument's name must match.

```
def show (name, age) :
    print(name, age)

show(name="Geekyshows", age=62)
```

```
def show (name, age) :
    print(name, age)

show(age=62, name="Geekyshows")
```

```
def show (name, age) :
    print(name, age)

show(name="Geekyshows", age=62, roll=101)
```

Note - Number of argument must be equal in formal and actual argument, Not more Not less

Default Arguments

Sometime we mention default value to the formal argument in function definition and we may not required to provide actual argument, In this case default argument will be used by formal argument.

If we do not provide actual argument for formal argument explicitly while calling the function then formal argument will use default value on the other hand if we provide actual argument then it will use provided value

```
def show (name, age=27) :
    print(name, age)

show(name="Geekyshows")
```

```
def show (name, age=27) :
    print(name, age)

show(name="Geekyshows", age=62)
```

Note - Number of argument must be equal in formal and actual argument, Not more Not less

Variable Length Arguments

Variable length argument is an argument that can accept any number of values.

The variable length argument is written with * symbol.

It stores all the value in a tuple.

```
def add (*num) :
    z = num[0]+num[1]+num[2]
    print(z)

add(5, 2, 4)
```

```
def add (x, *num) :
    z = x+num[0]+num[1]
    print(z)

add(5, 2, 4)
```

Keyword Variable Length Arguments

Keyword Variable length argument is an argument that can accept any number of values provided in the form of key-value pair.

The keyword variable length argument is written with ** symbol.

It stores all the value in a dictionary in the form of key-value pair.

```
def add (**num) :
    z = num['a']+num['b']+num['c']
    print(z)

add(a=5, b=2, c=4)
```

```
def add (x, **num) :
    z = x+num['a']+num['b']
    print(z)

add(3, a=5, b=2)
```

Local Variables

The variable which are declared inside a function called as Local Variable.

Local variable scope is limited only to that function where it is created. It means local variable value is available only in that function not outside of that function.

```
def add (y) :
    x = 10
    print(x)
    print(x + y)
add(20)
print(x)
```

Local Variable

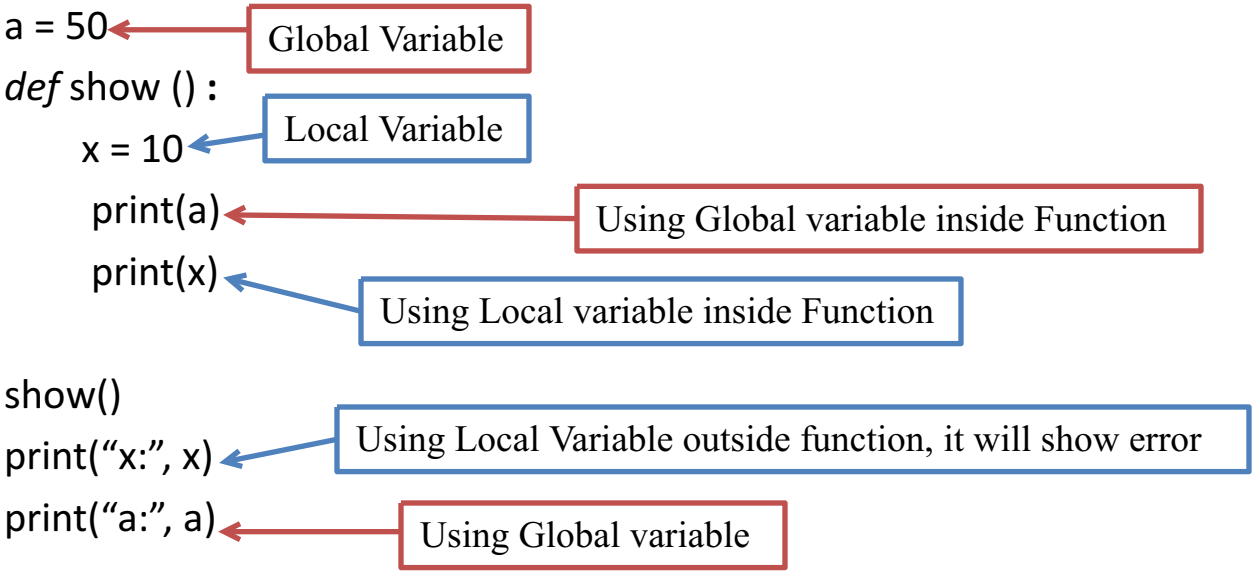
Using Local variable inside Function

Using Local Variable outside function, it will show error

Global Variables

When a variable is declared above a function, it becomes global variable. These variables are available to all the function which are written after it.

The scope of global variable is the entire program body written below it.



Global Keyword

If local variable and global variable has same name then the function by default refers to the local variable and ignores the global variable.

It means global variable is not accessible inside the function but possible to access outside of function.

In this situation, If we need to access global variable inside the function we can access it using global keyword followed by variable name.

Global Keyword

```
a = 50
def show () :
        a = 10
        print(a)
show()
print("a:", a)
```

```
a = 50
def show () :
        global a
        print(a)
        a = 20
        print(a)
show()
print("a:", a)
```

globals ( ) Function

This function returns a table of current global variables in the form of dictionary.

```
a = 50
def show () :
    a = 10
    print(“Local Variable A:”, a)
    x = globals()[‘a’]
    print(“X:”, x)
show()
print(“Global Variable A:”, a)
```

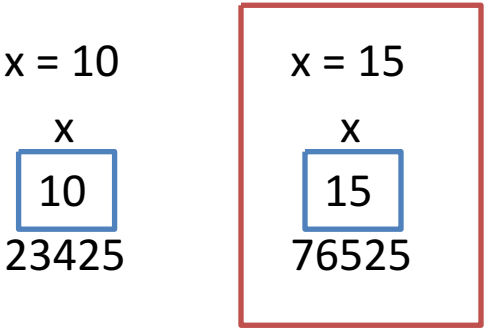
Pass/Call by Object Reference

In C, Java and some other languages we pass value to a function either by value or by reference widely known as “Pass by Value” and “Pass by Reference”.

In Python, Neither of these two concepts is applicable rather the values are sent to functions by means of object reference.

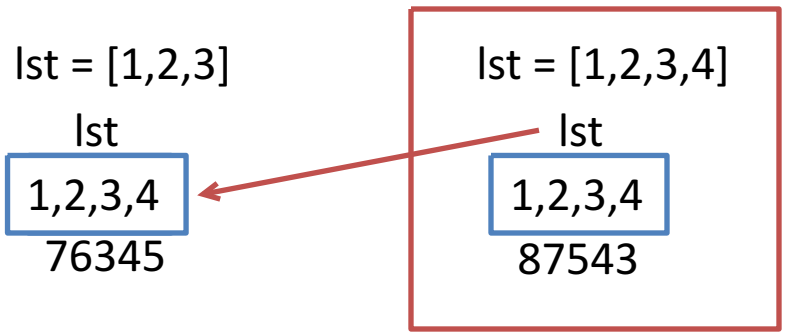
When we pass value like number, strings, tuples or lists to function, the references of these objects are passed to function.

```
def val(x):
    x = 15
    print(x, id(x))

x = 10
val(x)
print(x, id(x))
```



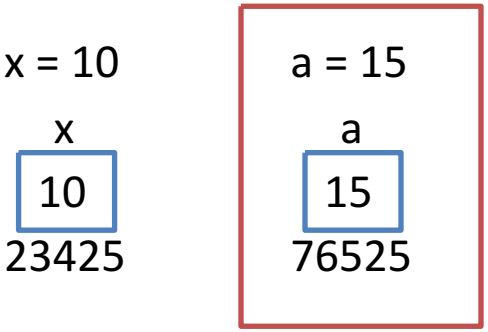
A new object is created in the memory because integer objects are immutable (not modifiable).

```
def val(lst):
    lst.append(4)
    print(lst, id(lst))

lst = [1, 2, 3]
print(lst, id(lst))
val(lst)
print(lst, id(lst))
```



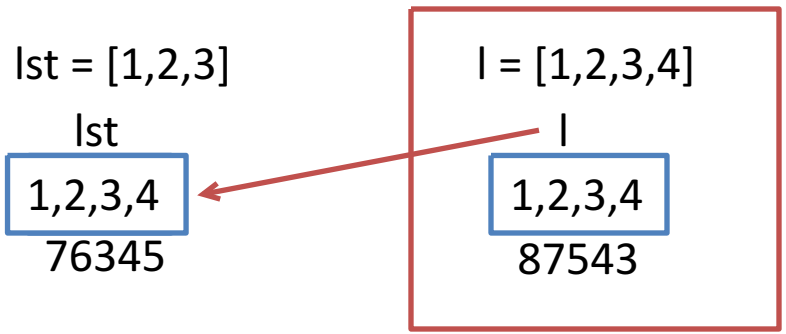
A new object is not created in the memory because list objects are mutable (modifiable). It simply add new element to the same object.

```
def val(a):
    a = 15
    print(a, id(a))

x = 10
val(x)
print(x, id(x))
```



A new object is created in the memory because integer objects are immutable (not modifiable).

```
def val(l):
    l.append(4)
    print(l, id(l))

lst = [1, 2, 3]
print(lst, id(lst))
val(lst)
print(lst, id(lst))
```



A new object is not created in the memory because list objects are mutable (modifiable). It simply add new element to the same object.

Pass/Call by Object Reference

In Python, values are passed to functions by object references.

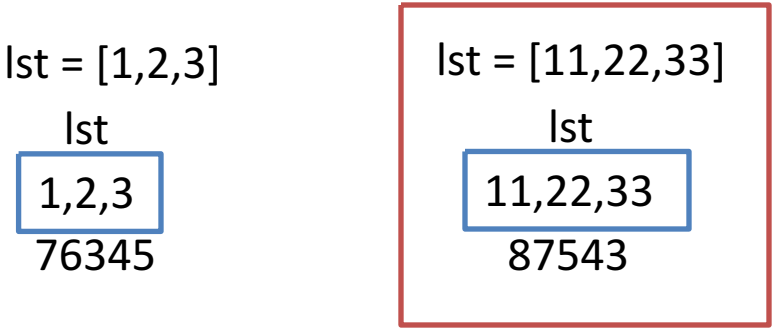
If object is immutable (not modifiable) then the modified value is not available outside the function.

If object is mutable (modifiable) then the modified value is available outside the function.

Immutable Objects – Integer, Float, String and Tuple

Mutable Objects – List and Dictionary

```
def val(lst):
    print(lst, id(lst))
    lst = [11, 22, 33]
    print(lst, id(lst))

lst = [1, 2, 3]
print(lst, id(lst))
val(lst)
print(lst, id(lst))
```



When we create a new object inside function then it will not be available outside function

Recursion

A function calling itself again and again to compute a value is referred to Recursive Function or Recursion.

Expression vs Statement

Expression/Expression Statements

Expression statements are used to compute and write a value, or to call a procedure.

Ex:- Operators like Addition, Subtraction, Function Call etc

Statement

Statements on the other hand, are everything that can make up a line or several lines of Python code. Expressions are also statements.

Ex:- if statement, assignment statement, loop

Anonymous Function or Lambdas

A function without a name is called as Anonymous Function. It is also known as Lambda Function.

Anonymous Function are not defined using *def* keyword rather they are defined using *lambda* keyword.

Syntax:-

lambda argument_list : expression

Ex:-

lambda x : print(x)

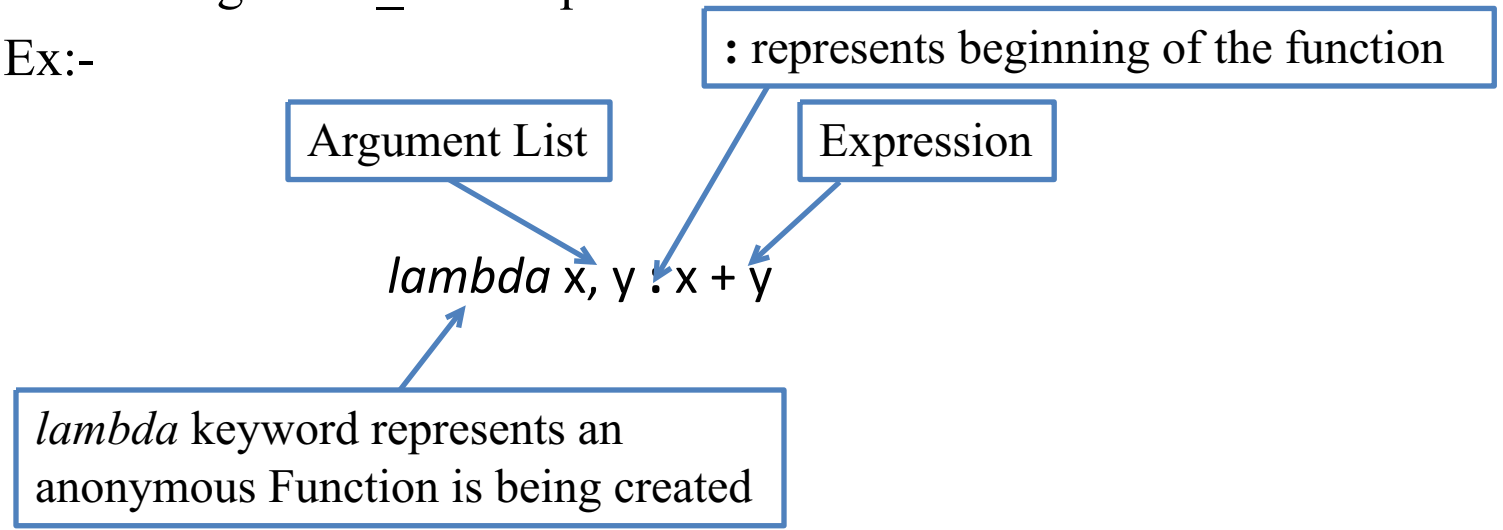
lambda x, y : x + y

Creating a Lambda Function

Syntax:-

lambda argument_list : expression

Ex:-



Calling Lambda Function

```
sum = lambda x : x + 1
sum(5)

 add = lambda x, y : x + y
add(5, 2)
```

Anonymous Function or Lambdas

- Lambda Function doesn't have any Name

  Ex:- *lambda* x : print(x)

- Lambda function returns a function

  Ex:- show = *lambda* x : print(x)

- Lambda function can take zero or any number of argument but contains only one expression

  Ex:- *lambda* x, y : x + y

- In lambda Function there is no need to write return statement
- It can only contain expressions and can't include statements in its body
- You can use all the type of Actual Arguments

Nested Lambda Function

When we write a lambda function inside another lambda function that is called nested lambda function.

```
add = lambda x=10 : (lambda y : x + y)
a = add()
print(a)
print(a(20))
```

Passing lambda Function to another Function

We can pass lambda function to another function.

```
def show(a):
        print(a(8))

show(lambda x: x)
```

Returning lambda Function

We can return a lambda function from function.

```
def add():
        y = 20
        return (lambda x : x+y)

a =add()
print(a(10))
```

Immediately Invoked Function Expressions (IIFE)

```
sum = lambda x : x + 1
sum(5)

(lambda x : x + 1)(5)

add = lambda x, y : x + y
add(5, 2)

(lambda x, y : x + y)(5, 2)
```

Function Decorator

A Decorator function is a function that accepts a function as parameter and returns a function.

A decorator takes the result of a function, modifies the result and returns it.

In Decorators, functions are taken as the argument into another function and then called inside the wrapper function.

We use @function_name to specify a decorator to be applied on another function.

Math Module

It is a module that contains several functions to perform mathematical operation.

If we want to use this module then we have to import it first.

from math import *

Math Functions

- floor(n) – Decrease n value to the previous integer value . If n is integer, then same value is returned.
- ceil(n) – Raise n value to the next higher integer. If n is integer, then same value is returned.
- fabs(n) – Returns absolute value or positive quantity of n
- factorial(n) – Returns factorial value of n
- sqrt(n) – Returns positive square root of n
- pow(n,m) – Returns n value to the power of m

Math Functions

- sin(n) – Returns the sine of n
- cos(n) – Returns the cosine of n
- tan(n) – Returns the tangent of n
- gcd(n, m) - Return the greatest common divisor of the integers a and b. If either a or b is nonzero, then the value of gcd(a, b) is the largest positive integer that divides both a and b. gcd(0, 0) returns 0.

https://docs.python.org/3/library/math.html

List

A list represents a group of elements.

Lists are very similar to array but there is major difference, an array can store only one type of elements whereas a list can store different type of elements.

Lists are mutable so we can modify it's element.

A list can store different types of elements which can be modified.

Lists are dynamic which means size is not fixed.

Lists are represented using square bracket [ ].

Ex:- a = [10, 20, -50, 21.3, 'Geekyshows']

Ex:- a = [10, 20, 50]

Creating a List

A list is similar to an array that consists of a group of elements or items.

Syntax:- list_name = [element1, element2, …..]

Ex:- a = [10, 20, -50, 21.3, ‘Geekyshows’]

Creating an Empty List

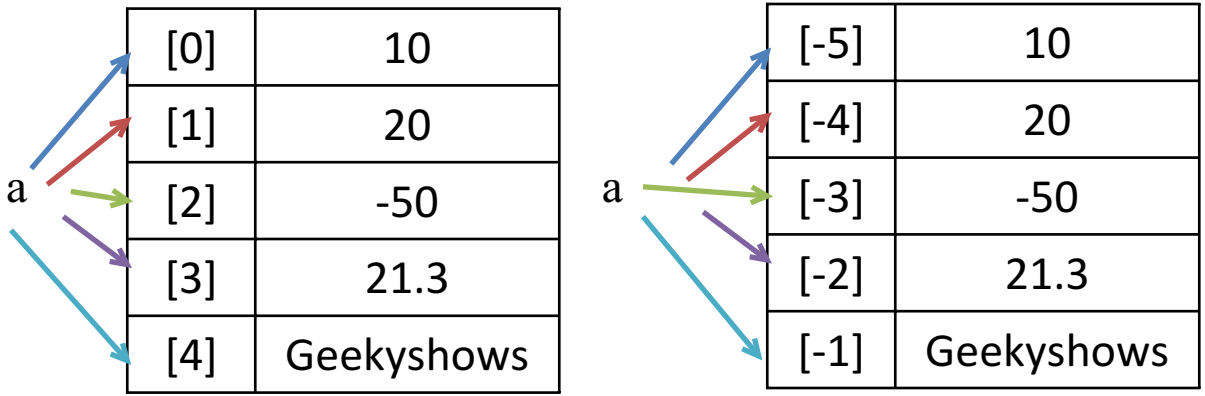
Syntax:- list_name = [ ]

Ex:- a = [ ]

Index

An index represents the position number of an list's element. The index start from 0 on wards and written inside square braces.

Ex:- a = [10, 20, -50, 21.3, 'Geekyshows']



Accessing List's Element

a = [10, 20, -50, 21.3, 'Geekyshows']

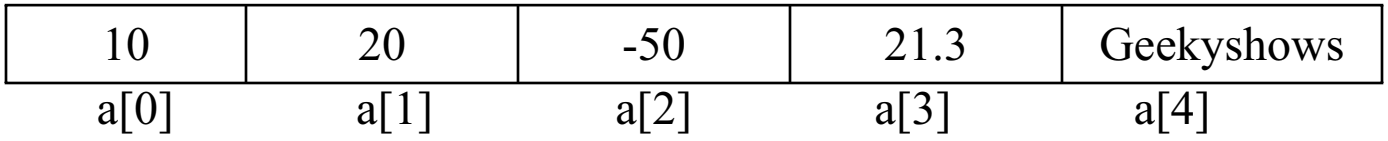
print(*a*[0])

print(*a*[1])

print(*a*[2])

print(*a*[3])

print(*a*[4])

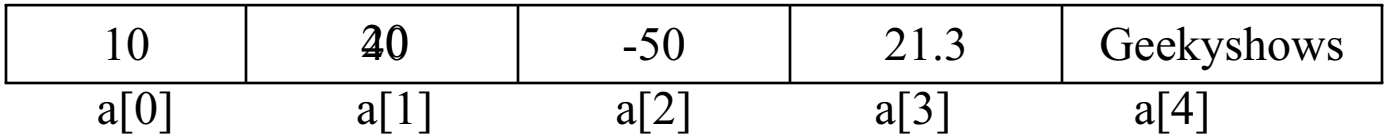


Modifying or Updating Element

Lists are mutable so we can modify it's element.

a = [10, 20, -50, 21.3, 'Geekyshows']

a[1] = 40



Accessing using for loop

a = [10, 20, -50, 21.3, ‘Geekyshows’]

Without index

```
for element in a:
        print(element)
```

With index

```
n = len(a)
for i in range(n):
        print(a[i])
```

Accessing using while loop

```
a = [10, 20, -50, 21.3, ‘Geekyshows’]

n = len(a)
i = 0
while i < n :
        print(a[i])
        i+=1
```

Deletion

del statement is used to delete an element of list or we can delete entire list using del statement.

a = [10, 20, -50, 21.3, ‘Geekyshows’]

Deleting Element

del a[2]

Deleting Entire List

del a

append ( )

This method is used to add an element at the end of the existing list.

Syntax:-

```
list_name.append(new_element)
```

Getting User input

```
a = []

n = int(input("Enter Number of Elements: "))
for i in range(n):
        a.append(int(input("Enter Element:")))

print("List:")
for element in a:
        print (element)
```

insert( )

This method is used to insert an element in a particular position of the existing list.

Syntax:-

```
list_name.insert(position_number, new_element)
```

pop ( )

This method is used to remove last element from the existing list.

Syntax:-

```
list_name.pop( )
```

pop (n)

This method is used to remove an element specified by position number, from the existing list and returns removed element.

Syntax:-

```
list_name.pop(position_number)
```

remove( )

This method is used to remove first occurrence of given element from the existing list. If it doesn’t found the element, shows valueError.

Syntax:-

```
list_name.remove(element)
```

index( )

This method returns position number of first occurrence of given element in the list. If it doesn’t found the element, shows valueError.

Syntax:-

```
list_name.index(element)
```

reverse ( )

This method is used to reverse the order of elements in the list.

Syntax:-

```
list_name.reverse( )
```

extend( )

This method is used to append another list or iterable object at the end of the list.

Syntax:-

list_name.extend(lst)

count( )

This method returns number of occurrence of a specified element in the list.

Syntax:-

list_name.count(specified_element)

sort( )

This method is used to sort the elements of the list into ascending order.

Syntax:-

```
list_name.sort()
```

clear( )

This method is used to delete all the elements from the list

Syntax:-

```
list_name.clear()
```

Slicing on List

Slicing on list can be used to retrieve a piece of the list that contains a group of elements. Slicing is useful to retrieve a range of elements.

Syntax:-

new_list_name = list_name[start:stop:stepsize]

List Concatenation

+ operator is used to do concatenation the list.

Ex:-

```
a = [10, 20, 30]
b = [1, 2, 3]
result = a + b
print(result)
```

Repetition of List

* Operator is used to repeat the elements of list.

Ex:-

```
b = [1, 2, 3]
result = b * 3
print(result)
```

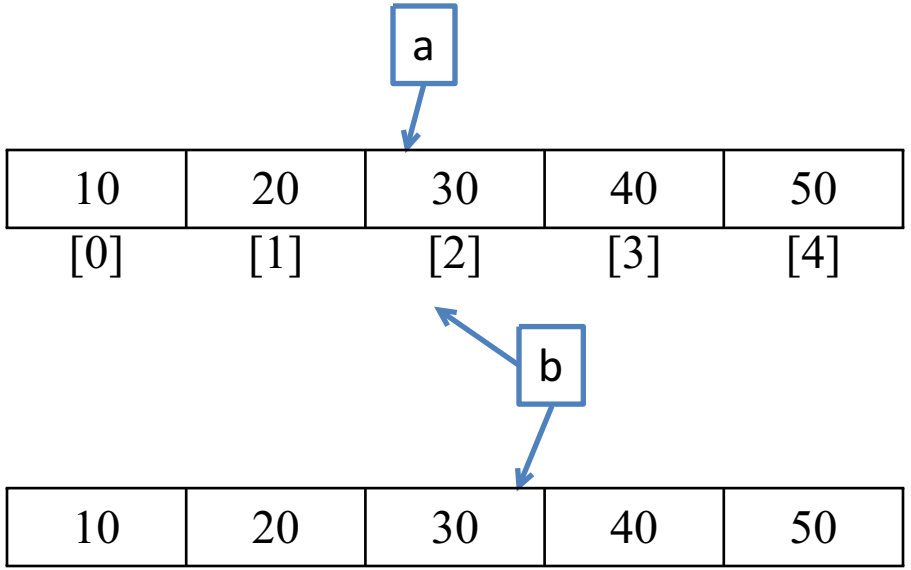
Aliasing List

Aliasing means giving another name to the existing object. It doesn't mean copying.

a = [10, 20, 30, 40, 50]

b = a

Modification in *a* will affect *b* and vice versa.



Copying List

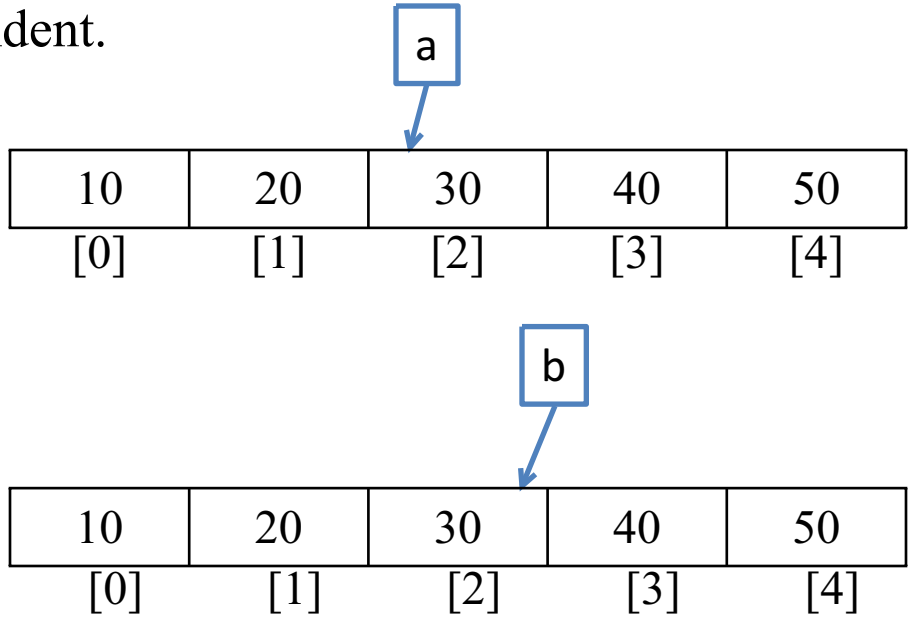
copy( ) method is used to copy all the elements of a list to another list.

When we copy a list a separate copy of all the elements is stored in another list. Both the list are independent.

a = [10, 20, 30, 40, 50]

b = a.copy()

Modification in *a* will not affect *b* and vice versa.



Cloning List

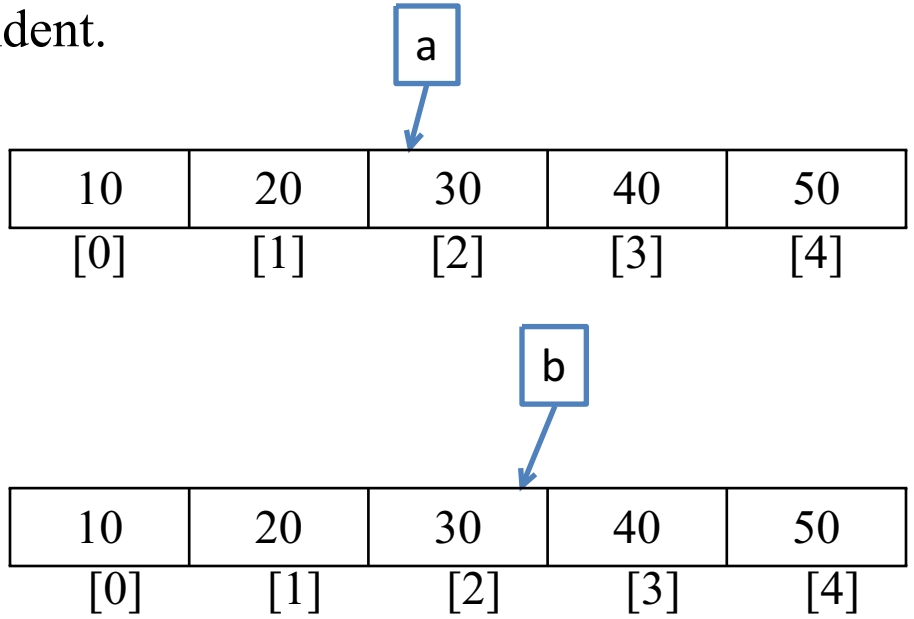
We can clone a list into another list using slicing.

When we clone a list a separate copy of all the elements is stored in another list. Both the list are independent.

a = [10, 20, 30, 40, 50]

b = a[:]

Modification in *a* will not affect *b* and vice versa.



Nested List

A list within another list is called as nested list or nesting of a list.

Ex:-

a = [10, 20, 30, [50, 60]]

b = [50, 60]

a = [10, 20, 30, b]

a = [[10, 20, 30], [40, 50, 60]]

a = [[10, 20, 30],

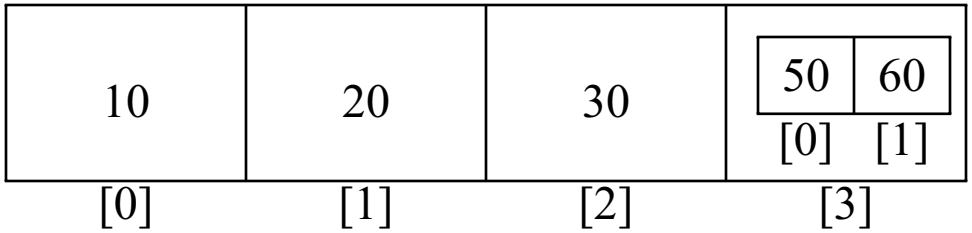
[40, 50, 60]]

Index

a = [10, 20, 30, [50, 60]]

b = [50, 60]

a = [10, 20, 30, b]



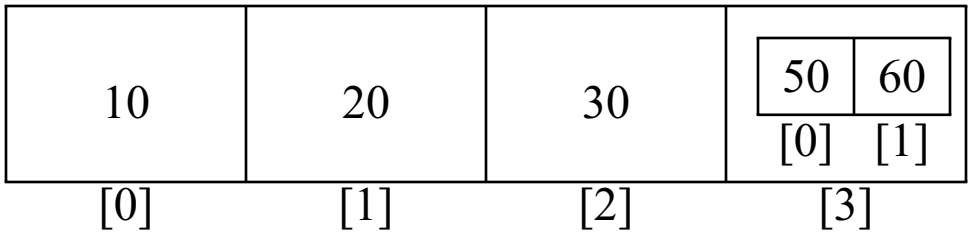
a = [[10, 20, 30], [40, 50, 60]]

a = [[10, 20, 30],
[40, 50, 60]]

	0	1	2
0	10	20	30
1	40	50	60

Accessing Nested List

a = [10, 20, 30, [50, 60]]



b = [50, 60]
a = [10, 20, 30, b]
print(a[0])
print(a[1])
print(a[2])
print(a[3][0])
print(a[3][1])

print(a) # All elements

Accessing Nested List

```
a = [  [10, 20, 30],
       [40, 50, 60]  ]
print(a[0][0])
print(a[0][1])
print(a[0][2])
print(a[1][0])
print(a[1][1])
print(a[1][2])

print(a)              # All elements
```

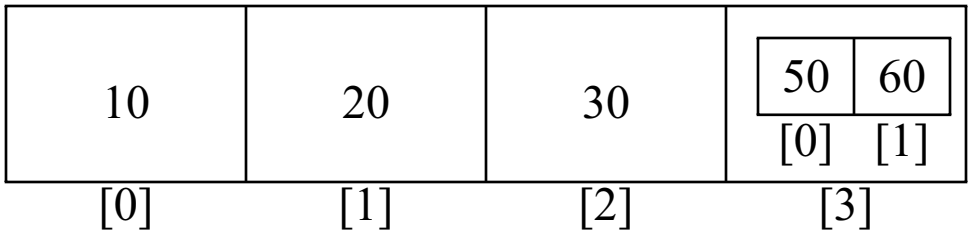
	0	1	2
0	10	20	30
1	40	50	60

Modifying Nested List

a = [10, 20, 30, [50, 60]]

b = [50, 60]

a = [10, 20, 30, b]



a[1] = 100

a[3][0] = 5

Modifying Nested List

a = [[10, 20, 30],
[40, 50, 60]]

	0	1	2
0	10	20	30
1	40	50	60

a[0][1] = 2

a[1][2] = 6

Accessing Nested list using for loop

```
a = [  [10, 20, 30],
       [40, 50, 60]  ]
```

Without index

```
for r in a:
    for c in r:
        print(c)
    print( )
```

Outer for loop

Inner for loop

With index

```
n = len(a)
for i in range(n):
    for j in range(len(a[i])):
        print(a[i][j])
    print ( )
```

Inner for loop

Outer for loop

The outer for loop represents the rows and the inner for loop represents the columns in each row.

Accessing Nested list using while loop

```
a = [  [10, 20, 30],
       [40, 50, 60]  ]
n = len(a)
i = 0
while i < n :
        j = 0
        while j < len(a[i]):
                print(a[i][j])
                j+=1
        i+=1
```

List ( ) Function

This is used to create a list. It returns a mutable list of elements.

Syntax:-

list() ← Creates Empty List

list(iterable_object) ← Creates a list of elements

Ex:-

list()

list(“GeekyShows”)

List ( ) Function

We can use list and range function to create a list.

Syntax:-

list(range(start,stop,stepsize))

Ex:-

list(range(0, 5))

Passing List to Function

We can pass a list to a function while calling function.

```
def show(l):
        print(l)
        print(type(l))
        for i in l:
                print(i)


lst = [10, 20, 30, 'GeekyShows']
show(lst)
```

Higher Order Function

filter ( ) Function – The filter function is used to filter out the elements of an iterable (sequence) depending on a function that tests each element in the sequence to be true or not.

It returns those elements of sequence, for which function is true.

Syntax:-

filter(function_name, iterable)

Function_name – It's name of a function which tests each element in the sequence return True or False. If function is None, returns the elements that are true.

iterable – Iterable may be either a sequence, list, string, tuple, a container which supports iteration, or an iterator.

Higher Order Function

map ( ) Function – This function executes a specified function on each element of the iterable (sequence) and perhaps changes the elements.

Syntax:-

map(function_name, iterable)

Function_name - It’s name of a function which perform an operation on all the elements of the sequence and modified elements are returned which can be stored in another sequence.

iterable – Iterable may be either a sequence, list, string, tuple, a container which supports iteration, or an iterator.

Higher Order Function

reduce ( ) Function – This function is used to reduce a sequence of elements to a single value by processing the elements according to a function supplied. It returns a single value.

This function is a part of functools module so you have to import it before using.

Syntax:-

from functools import reduce

reduce(function_name, sequence)

Generator

Generators are functions that return a sequence of values. We use yield statement to return the value from function.

Yield Statement

Yield statement returns the elements from a generator function into a generator object.

Ex:- yield a

next ( ) Function

This function is used to retrieve element by element from a generator object.

Syntax:- next(gen_obj)

Tuple

Tuple – A tuple contains a group of elements which can be same or different types.

Tuples are immutable.

It is similar to List but Tuples are read-only which means we can not modify it's element.

Tuples are used to store data which should not be modified.

It occupies less memory compare to list.

Tuples are represented using parentheses ( ).

Ex:- a = (10, 20, -50, 21.3, 'Geekyshows')

Creating Empty Tuple

Syntax:- tuple_name = ( )

Ex:- a = ( )

Creating Tuple

We can create tuple by writing elements separated by commas inside parentheses.

With one Element

b = (10) ← It will become integer

c = (10,)

With Multiple Elements

d = (10, 20, 30, 40)

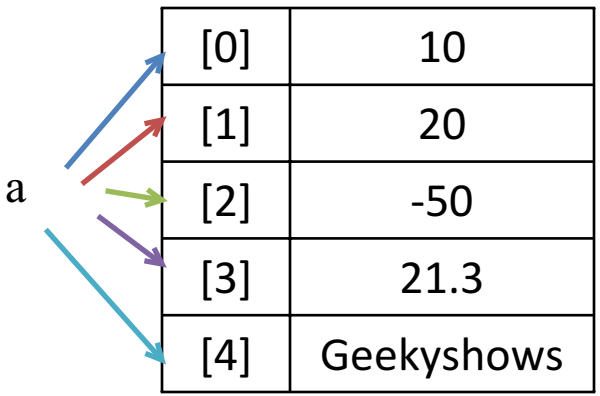
e = (10, 20, -50, 21.3, 'GeekyShows')

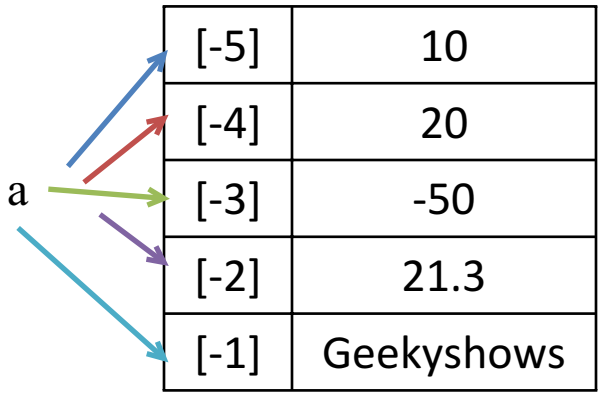
f = 10, 20, -50, 21.3, 'GeekyShows' ← It will become a tuple

Index

An index represents the position number of an tuple's element. The index start from 0 on wards and written inside square braces.

Ex:- a = (10, 20, -50, 21.3, 'Geekyshows')





Accessing Tuple's Element

a = (10, 20, -50, 21.3, 'Geekyshows')

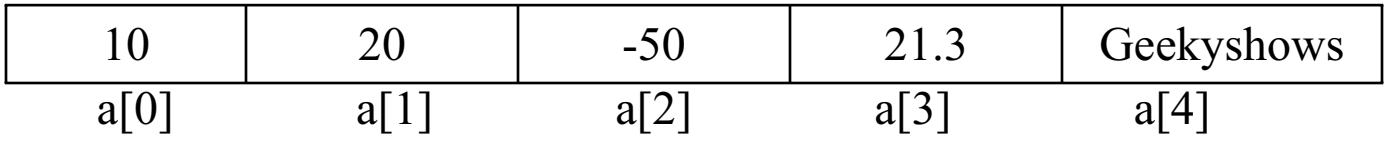
print(*a*[0])

print(*a*[1])

print(*a*[2])

print(*a*[3])

print(*a*[4])



Accessing using for loop

a = (10, 20, -50, 21.3, ‘Geekyshows’)

Without index

```
for element in a:
        print(element)
```

With index

```
n = len(a)
for i in range(n):
        print(a[i])
```

Accessing using while loop

```
a = (10, 20, -50, 21.3, ‘Geekyshows’)


n = len(a)
i = 0
while i < n :
        print(a[i])
        i+=1
```

Slicing on Tuple

Slicing on tuple can be used to retrieve a piece of the tuple that contains a group of elements. Slicing is useful to retrieve a range of elements.

Syntax:-

new_tuple_name = tuple_name[start:stop:stepsize]

Tuple Concatenation

+ operator is used to do concatenation the tuple.

Ex:-

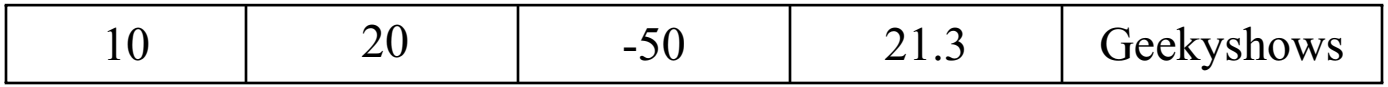
```
a = (10, 20, 30)
b = (1, 2, 3)
result = a + b
print(result)
```

Modifying Element

Tuples are immutable so it is not possible to modify, update or delete it's element.

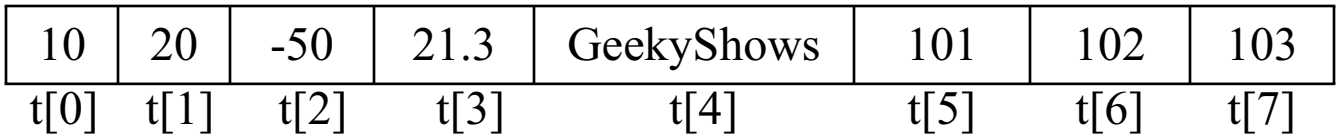
a = (10, 20, -50, 21.3, 'Geekyshows')

a[1] = 40



b = (101, 102, 103)

t = a + b



Deleting Tuple

You can delete entire tuple but not an element of tuple.

a = (10, 20, -50, 21.3, ‘Geekyshows’)

del a

10	20	-50	21.3	Geekyshows
a[0]	a[1]	a[2]	a[3]	a[4]

Repetition of Tuple

* Operator is used to repeat the elements of tuple.

Ex:-

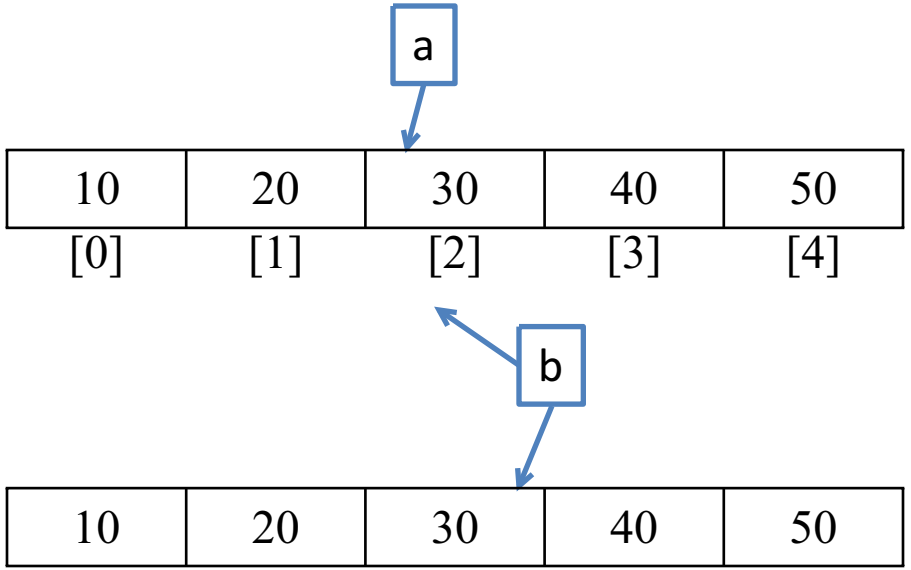
```
b = (1, 2, 3)
result = b * 3
print(result)
```

Aliasing Tuple

Aliasing means giving another name to the existing object. It doesn't mean copying.

a = (10, 20, 30, 40, 50)

b = a



Copying Tuple

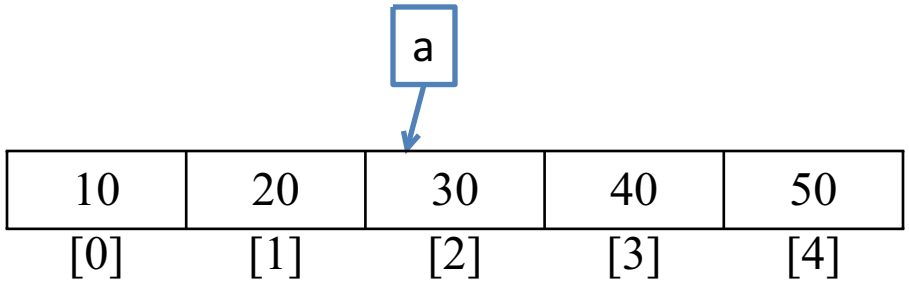
We can copy elements of tuple into another tuple using slice.

a = (10, 20, 30, 40, 50)

b = a

b = a[0:5]

b = a[1:4]



Nested Tuple

A tuple within another tuple is called as nested tuple or nesting of a tuple.

Ex:-

a = (10, 20, 30, (50, 60))

b = (50, 60)

a = (10, 20, 30, b)

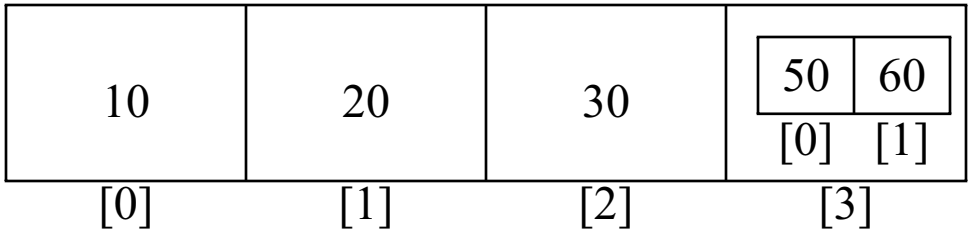
a = ((10, 20, 30), (40, 50, 60))

a = ((10, 20, 30),

(40, 50, 60))

Index

a = (10, 20, 30, (50, 60))



b = (50, 60)

a = (10, 20, 30, b)

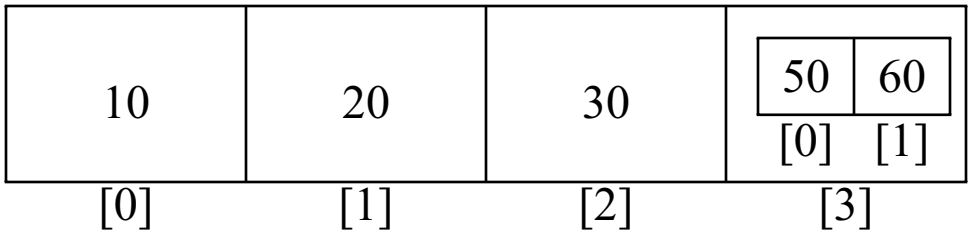
a = ((10, 20, 30), (40, 50, 60))

a = ((10, 20, 30),
(40, 50, 60))

	0	1	2
0	10	20	30
1	40	50	60

Accessing Nested Tuple

```
a = (10, 20, 30, (50, 60))

b = (50, 60)
a = (10, 20, 30, b)
print(a[0])
print(a[1])
print(a[2])
print(a[3][0])
print(a[3][1])

print(a)                # All elements
```



Accessing Nested Tuple

```
a = (  (10, 20, 30),
       (40, 50, 60)  )
print(a[0][0])
print(a[0][1])
print(a[0][2])
print(a[1][0])
print(a[1][1])
print(a[1][2])

print(a)              # All elements
```

	0	1	2
0	10	20	30
1	40	50	60

Accessing Nested tuple using for loop

a = ((10, 20, 30),
(40, 50, 60))

Without index

```
for r in a:
    for c in r:
        print(c)
    print( )
```

Outer for loop

Inner for loop

With index

```
n = len(a)
for i in range(n):
    for j in range(len(a[i])):
        print(a[i][j])
    print ( )
```

Inner for loop

Outer for loop

The outer for loop represents the rows and the inner for loop represents the columns in each row.

Accessing Nested tuple using while loop

```
a = (  (10, 20, 30),
       (40, 50, 60)  )
n = len(a)
i = 0
while i < n :
        j = 0
        while j < len(a[i]):
                print(a[i][j])
                j+=1
        i+=1
```

Passing Tuple to Function

We can pass a tuple to a function while calling function.

```
def show(t):
        print(t)
        print(type(t))
        for i in t:
                print(i)

tup = (10, 20, 30, 'GeekyShows‘)
show(tup)
```

List of Tuples

a = [10, 20, (30, 40)]

b = [(10, 20), (30, 40)]

Tuple of Lists

a = (10, 20, [30, 40])

b = ([10, 20], [30, 40])

Set Type

A set is an unordered collection of elements much like a set in mathematics.

The order of elements is not maintained in the sets. It means the elements may not appear in the same order as they are entered into the set.

A set does not accept duplicate elements.

Set is mutable so we can modify it.

Sets are unordered so we can not access its element using index.

Sets are represented using curly brackets { }

a = {10, 20, 30, “GeekyShows”, “Raj”, 40}

Creating a Set

A set is created by placing all the items (elements) inside curly braces {}, separated by comma. A set does not accept duplicate elements.

Elements can be of different types except mutable element, like list, set or dictionary.

Ex:-

a = {10, 20, 30}

a = {10, 20, 30, “GeekyShows”, “Raj”, 40}

a = {10, 20, 30, “GeekyShows”, “Raj”, 40, 10, 20}

Creating Empty Set

We can create an empty set using set( ) function.

a = set()

Accessing elements

Sets are unordered so we can not access its element using index.

a = {10, 20, “GeekyShows”, “Raj”, 40}

a[0]

print(a)

Modifying Elements

Sets are mutable but as we can not access elements using index so we can not modify it

Adding one Element

We can add a new element to set using add( ) method.

Syntax:-

set_name.add(new_element)

a.add(‘Python’)

Adding Multiple Elements

We can add multiple elements to set using update( ) method.

The update() method can take tuples, lists, strings or other sets as its argument.

Syntax:- set_name.update(elements)

Ex:-

a.update([101, 102, 103])

Deleting Element

We can delete element using remove ( ) or discard( ) methods.

Remove( ) method raise an error if element doesn't exists while discard( ) method remains unchanged.

Syntax:- set_name.remove(element)

set_name.discard(element)

Ex:-

a.remove('GeekyShows')

a.discard('GeekyShows')

Copying Elements

Copy ( ) Method is used to copy existing set's elements into another set.

Syntax:- new_set_name = set_name.copy()

Ex:- new_a = a.copy()

Clearing All Elements

Clear ( ) Method is used remove all elements to the set

Syntax:- set_name.clear()

Ex:- a.clear()

Some other Methods of set

- intersection ( )
- union ( )
- difference ( )
- issubset ( )
- issuperset ( )

Dictionary

A Dictionary represents a group of elements in the form of key value pairs.

Dictionary in Python is an unordered collection.

Dictionaries are mutable so we can modify it’s item, without changing their identity.

Dictionaries are represented using curly bracket { }.

Ex:-

stu = {101: ‘Rahul’, 102: ‘Raj’, 103: ‘Sonam’ }

fees= {‘rahul’:2000, ‘raj’:3000, ‘sonam’:8000}

Creating an Empty Dictionary

Syntax:- dict_name = { }

Ex:- fees = { }

Creating a Dictionary

A dictionary is created in the form of key-value pair where keys can't be repeated and must be immutable and values can be of any datatype and can be duplicated.

keys are case sensitive.

Syntax:- dict_name = {key1:value1, key2:value2,…}

Ex:-

```
stu = {101: 'Rahul', 102: 'Raj', 103: 'Sonam' }
fees = {'rahul':2000, 'raj':3000, 'sonam':8000 }
fees = {
        'rahul': 2000,
        'raj': 3000,
        'sonam': 8000
    }
```

Key Rules

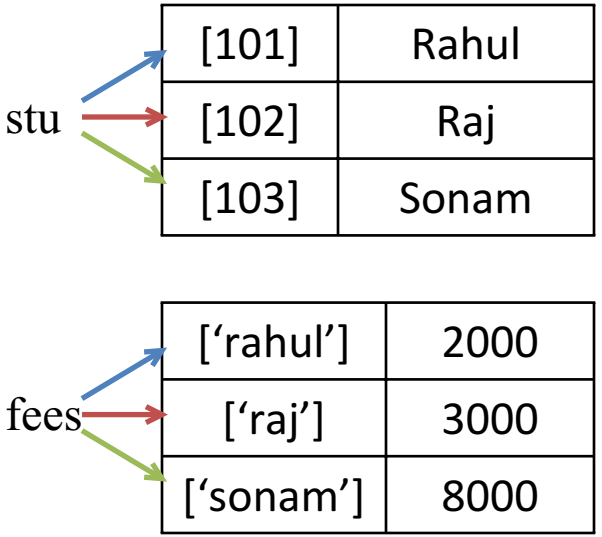
While writing key we must follow the following rules:

- Key should be unique.
- If we mention same key again, the old key will be overwritten.
- Key should be immutable type ex:- integer, string or tuple.
- We can not use list or dictionary as key.

Accessing Dictionary

We can access the value of a dictionary by referring to its key name, inside square brackets.

```
stu = {101: ‘Rahul’, 102: ‘Raj’, 103: ‘Sonam’ }
fees = {‘rahul’:2000, ‘raj’:3000, ‘sonam’:8000}
print(stu[101])
print(stu[102])
print(stu[103])

print(fees[‘rahul’])
print(fees[‘raj’])
print(fees[‘sonam’])
```



Modifying

We can modify the existing value of key by assigning a new value.

stu = {101: ‘Rahul’, 102: ‘Raj’, 103: ‘Sonam’ }

stu[102] = ‘Python’

Inserting/Adding new item

We can add an item to dictionary just by mentioning a new key-value pair into an existing dictionary.

If we mention a key which is already exists in the dictionary then the value gets updated/modified rather then adding a new item.

The new item my be added at any place in the dictionary as dictionary is an unordered collection.

stu = {101: ‘Rahul’, 102: ‘Raj’, 103: ‘Sonam’ }

stu[104] = ‘Geekyshows’

Deletion

We can delete an item of dictionary or entire dictionary using del statement.

stu = {101: ‘Rahul’, 102: ‘Raj’, 103: ‘Sonam’ }

Deleting an item

del stu[102]

Deleting entire Dictionary

del stu

Testing Key

We can check whether a key is already exists in the dictionary or not, for this purpose we use membership operator.

stu = {101: ‘Rahul’, 102: ‘Raj’, 103: ‘Sonam’ }

```
101 in stu          # If exists returns True
104 not in stu      # If not exists returns True
```

Clear ( ) Method

This method is used to remove all the elements from the dictionary.

Syntax:- dict.clear( )

Ex:- stu.clear( )

copy ( ) Method

This method is used to copy all the elements from the existing dictionary into a new dictionary.

Syntax:- dict.copy( )

Ex:-

new_stu = stu.copy( )

fromkeys ( ) Method

This method is used to create a new dictionary with the specified keys and values.

Syntax:- dict.fromkeys(keys, value)

Ex:-

key = (101, 102, 103)

value = ‘GeekyShows’

new_stu = dict.fromkeys(key, value)

get ( ) Method

This method returns the value of the specified key.

If key is not found then it will return none or default value.

Syntax:- dict_name.get(key, defaultvalue)

Ex:-

stu.get(104)

stu.get(104, ‘GeekyShows’)

items ( ) Method

This method returns an object that contains key-value pairs of dictionary.

The pairs are stored as tuples in the object.

Syntax:- dict_name.items()

Ex:-

stu.items()

keys ( ) Method

This method returns a sequence of keys from the dictionary .

Syntax:- dict_name.keys()

Ex:- stu.keys()

values ( ) Method

This method returns a sequence of values from the dictionary .

Syntax:- dict_name.values()

Ex:- stu.values()

update ( ) Method

This method is used to update the dictionary with the specified key value pair.

Syntax:- dict_name.update(iterable)

Ex:- stu.update({105: ‘Gupchup’})

pop ( ) Method

This method is used to remove the item with specified key.

It returns the removed item’s value.

If key is not found then the a default value is returned.

If key is not found and default value is not given then shows KeyError.

Syntax:- dict_name.pop(key, defaultvalue)

Ex:- stu.pop(101)

Ex:- stu.pop(110, ‘GeekyShows’)

popitem ( ) Method

This method is used to remove the item which was last inserted into the dictionary.

It returns the removed item in the form of tuple, Pairs are returned in LIFO order.

Syntax:- dict_name.popitem( )

Ex:- stu.popitem()

setdefault ( ) Method

This method returns the value of the specified key.

If key is not found then it inserts key with the specified value.

Syntax:- dict_name.setdefault(key, value)

Ex:- stu.setdefault(109, ‘Rohit’)

Nested Dictionary

A dictionary within another dictionary is called as nested dictionary or nesting of a dictionary.

```
nested_dict = { 'dict1': {'key_A': 'value_A'},
                'dict2': {'key_B': 'value_B'} }
```

Creating Empty Nested Dict

Syntax:-

```
nested_dict = { 'dict1': { },
                'dict2': { } }
```

Ex:-

```
a = {1: { },
     2: { } }
```

Creating Nested Dict

Syntax:-

nested_dict = { 'key_A': 'value_A', 'dict1': {'key_A': 'value_A'} }

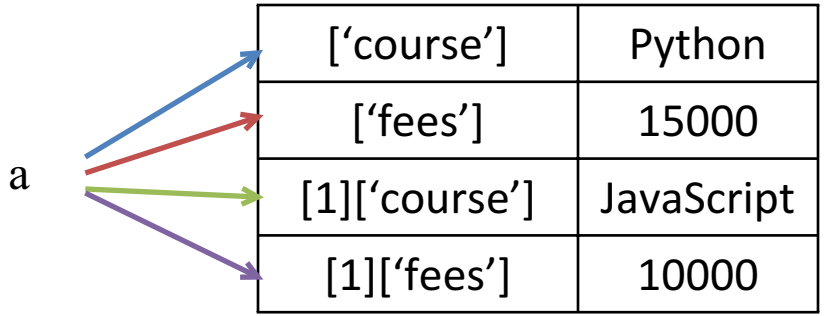
Ex:-

a = {‘course’: ‘Python’, ‘fees’:15000, 1: {‘course’: ‘JavaScript’, ‘fees’: 10000 } }

Accessing Dictionary

a = {‘course’: ‘Python’, ‘fees’:15000, 1: {‘course’: ‘JavaScript’, ‘fees’: 10000 } }

```
print(a[‘course’])
print(a[‘fees’])

print(a[1][‘course’])
print(a[1][‘fees’])
```



Modifying Nested Dict

a = {‘course’: ‘Python’, ‘fees’:15000, 1: {‘course’: ‘JavaScript’, ‘fees’: 10000 } }

a[‘course’] = ‘Machine Learning’

a[1][‘fees’] = 200000

Creating Nested Dict

Syntax:-

```
nested_dict = { 'dict1': {'key_A': 'value_A'},
                'dict2': {'key_B': 'value_B'} }
```

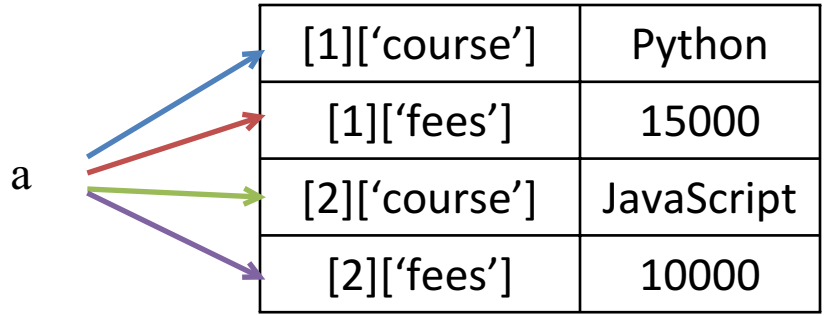
Ex:-

```
a = {1: {‘course’: ‘Python’, ‘fees’:15000},
     2: {‘course’: ‘JavaScript’, ‘fees’: 10000 } }
```

Accessing Dictionary

```
a = {1: {'course': 'Python', 'fees':15000},
     2: {'course': 'JavaScript', 'fees': 10000 } }
```

```
print(a[1]['course'])
print(a[1]['fees'])

print(a[2]['course'])
print(a[2]['fees'])
```



Modifying Nested Dict

```
a = {1: {‘course’: ‘Python’, ‘fees’:15000},
     2: {‘course’: ‘JavaScript’, ‘fees’: 10000 } }
a[1][‘course’] = ‘Machine Learning’
a[2][‘fees’] = 200000
```

Comprehension

Comprehension contains very compact code usually a single statement that perform a task.

- List Comprehension
- Set Comprehension
- Dictionary Comprehension

List Comprehension

List comprehension represents creation of new list from an iterable object that satisfy a given condition.

Syntax:-

new_list = [expression *for* item *in* iterable_object if_statement]

There can be zero or more If Statements.

There can be one or multiple for loops.

Ex:-

```
lst1 = [i+1 for i in range(20)]
lst2 = [i for i in range(20) if i%2==0]
lst3 = [i for i in range(11) if i%2==0 if i%3==0]
```

List Comprehension

- **List Comprehension with If else Statement**

Syntax:-

new_list = [expression if_statement else expression *for* item *in* iterable_object]

Ex:-

new_list = [i *if* i%2==0 *else* “Invalid” *for* i *in* range(10)]

Nested List Comprehension

```
lst =[ [i*j for j in range(4,7)] for i in range(6,8) ]
```

Set Comprehension

Set comprehension represents creation of new set from an iterable object that satisfy a given condition.

Syntax:-

new_set = {expression *for* item *in* iterable_object if_statement}

There can be zero or more If Statements.

There can be one or multiple for loops.

Ex:-

```
set1 = {i+1 for i in range(20)}
set2 = {i for i in range(20) if i%2==0}
set3 = {i for i in range(11) if i%2==0 if i%3==0}
```

Set Comprehension

- **Set Comprehension with If else Statement**

Syntax:-

new_set= {expression if_statement else expression *for* item *in* iterable_object}

Ex:-

new_set = {i *if* i%2==0 *else i*1000 for* i *in* range(10)}

Nested Set Comprehension

```
st = {(I, j) for j in range(4,7) for i in range(6,8)}
```

Dictionary Comprehension

Dictionary comprehension represents creation of new Dictionary from an iterable object that satisfy a given condition.

Syntax:-

new_dict = {expression(variable):expression(variable) *for* variable,variable *in* iterable_object if_statement}

There can be zero or more If Statements.

There can be one or multiple for loops.

Ex:-

```
dict1 = {k:v for k, v in lst}

dict2 = {n:n*2 for n in range(10)}

dict3 = {n:n*2 for n in range(20) if n%2==0}

dict4 = {n:n*2 for n in range(20) if n%2==0 if n%3==0}
```

Dictionary Comprehension

- **Comprehension with If else Statement**

Syntax:-

new_dict = {expression(variable): (expression if_statement else expression) *for* variable *in* iterable_object}

Ex:-

new_dict = {i :(“even” *if* i%2==0 *else* “Invalid” *for* i *in* range(10)}

When and Why

- Array
- List
- Tuple
- Set
- Dictionary

Array

Array is beneficial if you need to store group of elements of same datatype.

Array uses less memory than list

List

List is beneficial if you need to store group of elements of same or different datatype.

Lists are mutable so we can modify it’s element.

List uses more memory than Array.

Tuple

Tuple is beneficial if you need to store group of elements of same or different datatype.

Tuples are immutable so we can not modify it’s element.

Tuples are used to store data which should not be modified.

It occupies less memory compare to list.

Set

Set is beneficial if you need to store unordered group of elements of same or different datatype.

A set does not accept duplicate elements.

Sets are mutable so we can modify it’s element.

Sets are unordered so we can not access its element using index.

Dictionary

Dictionary is beneficial if you need to store elements in the form of key-value pair of same or different datatype.

Dictionary in Python is an unordered collection.

Dictionaries are mutable so we can modify it’s item.

id ( ) Function

This function returns the “identity” of an object.

The identity number is an integer which is guaranteed to be unique and constant for this object during its lifetime.

Two objects with non-overlapping lifetimes may have the same id() value.

CPython implementation detail – This is the address of the object in memory

Syntax:- id(object)

Ex:- id(lst)

type ( ) Function

This function returns type of the object.

Syntax:- type(object)

type(a)

getsizeof( ) Function

This function returns the size of an object in bytes.

The object can be any type of object.

All built-in objects will return correct results, but this does not have to hold true for third-party extensions as it is implementation specific.

Only the memory consumption directly attributed to the object is accounted for, not the memory consumption of objects it refers to.

This is part of sys module so you have to import sys module before using this function.

Syntax:-

from sys import *

getsizeof(object)

Ex:- from sys import *

getsizeof(a)

int ( ) Function

This function returns an integer object or return 0 if no arguments are given.

Syntax:- int(a)

float ( ) Function

This function returns an floating point number object.

Syntax:- float(a)

str ( ) Function

This function returns str version of object.

Syntax:- str(a)

list( ) Function

Rather than being a function, list is actually a mutable sequence type. This can be used in type casting to convert iterable to list.

Syntax:- list(iterable)

tuple( ) Function

Rather than being a function, tuple is actually an immutable sequence type. This can be also used in type casting to convert iterable to tuple.

Syntax:- tuple(iterable)

set( ) Function

This function returns a new set object, optionally with elements taken from iterable. This can be also used in type casting to convert iterable to set.

Syntax:- set(iterable)

dict( ) Function

This function creates a new dictionary. This can be also used in type casting to convert iterable to dict.

Syntax:- dict(**kwarg)

len ( ) Function

This function returns the length (the number of items) of an object.

The argument may be a sequence (such as a string, bytes, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set).

Syntax:- len(arg)

Ex:- len(lst)

min ( ) Function

This function returns the smallest item in an iterable or the smallest of two or more arguments.

If one positional argument is provided, it should be an iterable. The smallest item in the iterable is returned. If two or more positional arguments are provided, the smallest of the positional arguments is returned.

Syntax:-

min(iterable)

min(arg1, arg2,….)

Ex:- min(lst)

max ( ) Function

This function returns the largest item in an iterable or the largest of two or more arguments.

If one positional argument is provided, it should be an iterable. The largest item in the iterable is returned. If two or more positional arguments are provided, the largest of the positional arguments is returned.

Syntax:-

max(iterable)

max(arg1, arg2,….)

Ex:- max(lst)

sorted ( ) Function

This function returns a new sorted list from the items in iterable.

Syntax:- sorted(iterable)

Ex:- sorted(lst)

What Next ?

- Learn Advance Python
  https://www.youtube.com/playlist?list=PLbGui_ZYuhijd1hUF2VWiKt8FHNBa7kGb
- Always update yourself via Official Python Doc
  https://docs.python.org/3/tutorial/
- GeekyShows YouTube Channel
- Visit out official Website www.geekyshows.com
- Small Projects like Calculator