UNIT-1

TOPICS:-

Introduction to Python:-

1. Features of Python
2. Data types
3. Operators
4. Input and output
5. Control Statements.

Strings:-

1. Creating strings and basic operations on strings
2. String testing methods.
3. Lists
4. Dictionaries
5. Tuples

Topic 1.1.1- Features of Python

What is Python?

Python is a widely used popular programming language developed by **Guido Van Rossum.** The implementation of Python was started in December 1989 by Guido Van Rossum at CWI in Netherland. In February 1991, python was released.

Python is a popular general-purpose programming language that can be used for a wide variety of applications such as

- Machine Learning
- Artificial Intelligence
- Web Development
- Mobile Application
- Desktop GUI Application
- Scientific Calculation
- Numeric Calculation
- Software development
- Image Processing
- 3D CAD(Computer Aided Design)
- Audio or Video-based Applications
- Business, etc
- Deep learning

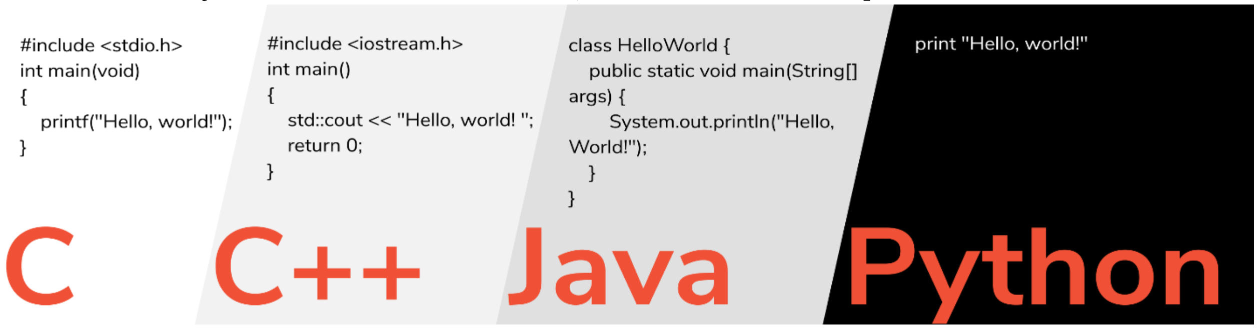
Who uses python?
Youtube – python used as scripting as well as programming language
Google – web search system
DropBox
Rasberry pi

Bit Torrent
NASA – National Aeronautics and Space Administration- Scientific Programming tasks
NSA – National Security Agency – Cryptography and Intellegence Analysis
CIA – Central Intelligence Agency
Facebook
Instagram
Netflix

Features of python:-

1. Simple and easy to learn – It resembles like English language a lot – It is easy to read and write python programs compared with other programming languages.
2. Free and Open Source – python is an example of FLOSS(Free/Libre and Open Source Software), which means ,
   Firstly, Python is **freely available for everyone**. You can download it from the Python Official Website www.python.org.

   Secondly, it is **open-source**. This means that its source code is available to the public. You can download it, change it, use it, and distribute it.
   It has a large community across the world that is dedicatedly working towards make new python modules and functions. Anyone can contribute to the Python community. The open-source means, "Anyone can download its source code without paying any amount."
3. High Level Language – python program is easier to read as well as code.
4. Expressive Language - Python can perform complex tasks using a few lines of code. A simple example, the hello world program you simply type **print("Hello World")**. It will take only one line to execute, while Ja va or C takes multiple lines.



```
#include <stdio.h>
int main(void)
{
   printf("Hello, world!");
}
```

```
#include <iostream.h>
int main()
{
   std::cout << "Hello, world! ";
   return 0;
}
```

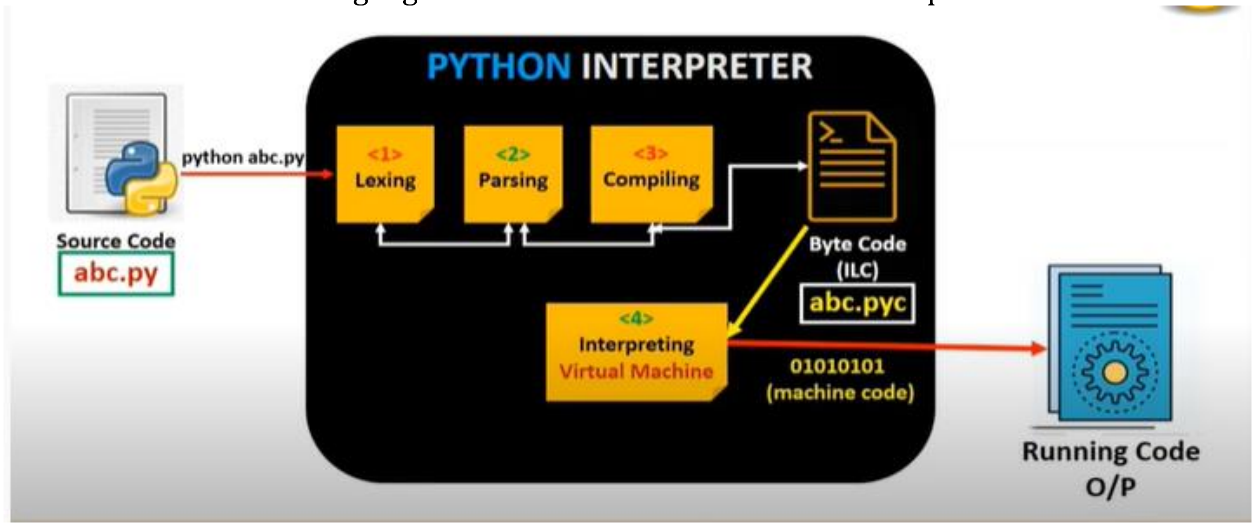
```
class HelloWorld {
   public static void main(String[]
args) {
      System.out.println("Hello,
World!");
   }
}
```

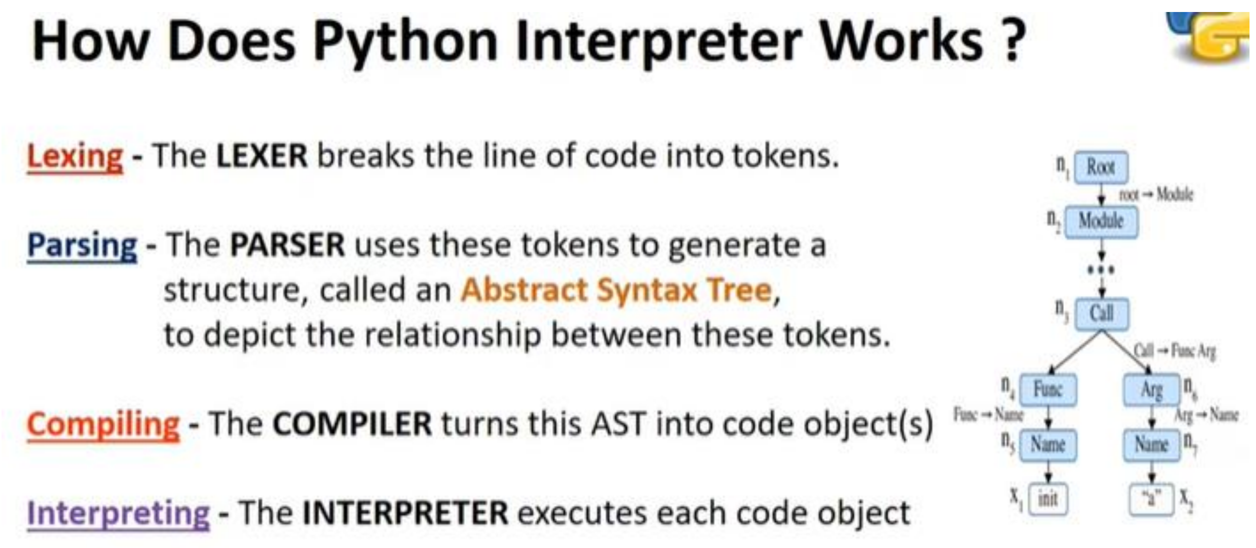
```
print "Hello, world!"
```

5. Portable - Python language is also a portable language. For example, if we have python code for windows and if we want to run this code on other platforms such as Linux, Unix, and Mac then we do not need to change it, we can run this code on any platform.
6. Supports different programming paradigm – Python supports procedure-oriented as well as object-oriented programming. It also supports multiple inheritance, unlike Java.

7. Object Oriented Programming - A programming language that can model the real world is said to be object-oriented. It focuses on objects and combines data and functions. Contrarily, a procedure-oriented language revolves around functions, which are code that can be reused.
8. Extensible – python code can invoke c and c++ libraries, can integrate with java and dotnet components.
9. Highly Dynamic – It means that the data type of a variable is decided at the run time and not in advance. Due to the presence of this feature, we do not need to specify the type of the variable during coding, thus saving time and increasing efficiency.
10. GUI Programming Support - Graphical User interfaces can be made using a module such as PyQt5, PyQt4, wxPython, or Tk in python.
    PyQt5 is the most popular option for creating graphical apps with Python.
11. Integrated Language - It can be easily integrated with languages like C, C++, JAVA etc.
12. Interpreted Language – Python is an interpreted language. An interpreter in general works very different from a compiler. An interpreter executes a code line by line and hence it gets easy for a programmer to debug errors.

    Also, if you have observed, even though your program has multiple errors, Python displays only one error at a time. Whereas a compiler compiles the entire code at once and displays a list of errors.
    When you run the program, python converts the source code to an intermediate form names byte codes and then converts to the binary language which is understandable to the computer.
    On the other hand running the programs in C and C++ needs a complier to convert the source code to the language which is understandable to the computer.





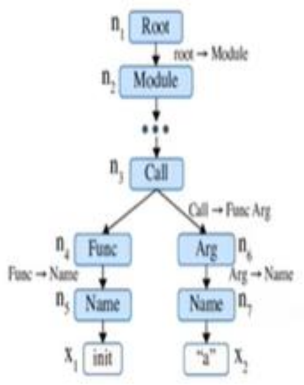
How Does Python Interpreter Works ?

Lexing - The **LEXER** breaks the line of code into tokens.

Parsing - The **PARSER** uses these tokens to generate a structure, called an **Abstract Syntax Tree**, to depict the relationship between these tokens.

Compiling - The **COMPILER** turns this AST into code object(s)

Interpreting - The **INTERPRETER** executes each code object



13. Large Standard Library - Python has a large standard library and this helps save the programmers time as you don't have to write your own code for every single logic. There are many libraries present in python for such as
    numpy, pandas, matplot lib – data science
    regular expression
    ns, unit-testing, web browsers, etc.

---***---

<u>Python Tokens</u>

Python Identifiers

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language. Thus, Manpower and manpower are two different identifiers in Python.

Reserved Words

The following list shows the Python keywords. These are reserved words and you cannot use them as constants or variables or any other identifier names. All the Python keywords contain lowercase letters only.

and	exec	Not
as	finally	or
assert	for	pass
break	from	print
class	global	raise
continue	if	return
def	import	try
del	in	while
elif	is	with
else	lambda	yield
except		

Lines and Indentation

Python does not use braces({ }) to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced. The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount.

Ex:-

```
if True:
    print ("True")
else:
  print ("False")
```

However, the following block generates an error-

```
if True:
    print ("Answer")
    print ("True")
else:
    print "(Answer")
  print ("False")
```

Quotation in Python

Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.
The triple quotes are used to span the string across multiple lines. For example, all the

following are legal

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

Comments in Python

A hash sign (#) that is not inside a string literal is the beginning of a comment. All characters after the #, up to the end of the physical line, are part of the comment and the Python interpreter ignores them.

Ex:-

#!/usr/bin/python3

```
# First comment
print ("Hello, Python!") # second comment
```

This produces the following result-

```
Hello, Python!
```

You can type a comment on the same line after a statement or expression-

```
name = "Madisetti" # This is again comment
```

Python does not have multiple-line commenting feature. You have to comment each line individually as follows-

```
# This is a comment.
# This is a comment, too.
# This is a comment, too.
# I said that already.
```

Variable and Data Types:-

Variables are nothing but reserved memory locations to store values. It means that when you create a variable, you reserve some space in the memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to the variables, you can store integers, decimals or characters in these variables.

Assigning Values to Variables

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable. For example-

```
#!/usr/bin/python3
counter = 100          # An integer assignment
miles   = 1000.0       # A floating point
name    = "John"       # A string
print (counter)
print (miles)
print (name)
```

Here, 100, 1000.0 and "John" are the values assigned to counter, miles, and name variables, respectively. This produces the following result –

```
100
1000.0
John
```

For example-

```
      a, b, c = 1, 2, "john"
```

Here, two integer objects with values 1 and 2 are assigned to the variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

Multiple Assignment

Python allows you to assign a single value to several variables simultaneously.

For example-

```
a = b = c = 1
```

Here, an integer object is created with the value 1, and all the three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables.

-------**--------**

Topic 1.1.2- Data types

Standard Data Types

- Numbers
- Strings
- Boolean
- None
- List
- Tuple
- Dictionary
- Set

Numbers:-

Python includes three numeric types to represent numbers: integers, float, and complex number.

In Python, integers are zero, positive or negative whole numbers without a fractional part and having unlimited precision, e.g. 0, 100, -10.

```
>>> 0
0
>>> 100
100
>>> -10
10
>>> 1234567890
1234567890
>>> y=5000000000000000000000000000000000000000000000000000
5000000000000000000000000000000000000000000000000000
```

Integers can be binary, octal, and hexadecimal values.

```
>>> 0b11011000 # binary
216
>>> 0o12 # octal
10
>>> 0x12 # hexadecimal
15
```

All integer literals or variables are objects of the int class. Use the type( ) method to get the class name, as shown below.

```
>>>type(100)
```

```
<class 'int'> # type of x is int


>>> x=1234567890
>>> type(x)
<class 'int'> # type of x is int

>>> y=5000000000000000000000000000000000000000000000000000
>>> type(y) # type of y is int
<class 'int'>
```

Note that integers must be without a fractional part (decimal point). It it includes a fractional then it becomes a float.

```
>>> x=5
>>> type(x)
<class 'int'>
>>> x=5.0
>>> type(x)
<class 'float'>
```

The `int( )` function converts a string or float to int.

```
>>> int('100')
100
>>> int('-10')
-10
>>> int('5.5')
5
>>> int('100', 2)
4
```

Binary

A number having 0b with eight digits in the combination of 0 and 1 represent the binary numbers in Python. For example, 0b11011000 is a binary number equivalent to integer 216.

```
>>> x=0b11011000
>>> x
216
>>> x=0b_1101_1000
>>> x
216

>>> type(x)
```

```
<class 'int'>
```

Octal

A number having 0o or 0O as prefix represents an octal number. For example, 0O12 is equivalent to integer 10.

```
>>> x=0o12
>>> x
10
>>> type(x)
<class 'int'>
```

Hexadecimal

A number with 0x or 0X as prefix represents hexadecimal number. For example, 0x12 is equivalent to integer 18.

```
>>> x=0x12
>>> x
18
>>> type(x)
<class 'int'>
```

Float

In Python, floating point numbers (float) are positive and negative real numbers with a fractional part denoted by the decimal symbol . or the scientific notation E or e, e.g. 1234.56, 3.142, -1.55, 0.23.

```
>>> f=1.2
>>> f
1.2
>>> type(f)
<class 'float'>
```

Floats has the maximum size depends on your system. The float beyond its maximum size referred as "inf", "Inf", "INFINITY", or "infinity". Float 2e400 will be considered as infinity for most systems.

```
>>> f=2e400
>>> f
inf
```

Scientific notation is used as a short representation to express floats having many digits. For example: 345.56789 is represented as 3.4556789e2 or 3.4556789E2

```
>>> f=1e3
>>> f
1000.0
>>> f=1e5
>>> f
100000.0
>>> f=3.4556789e2
>>> f
345.56789
```

Use the float( ) function to convert string, int to float.

```
>>> float('5.5')
5.5
>>> float('5')
5.0
>>> float('     -5')
-5.0
>>> float('1e3')
1000.0
>>> float('-Infinity')
-inf
>>> float('inf')
inf
```

Complex Number

A complex number is a number with real and imaginary components. For example, 5 + 6j is a complex number where 5 is the real component and 6 multiplied by j is an imaginary component.

```
>>> a=5+2j
>>> a
(5+2j)

>>> type(a)
<class 'complex'>
```

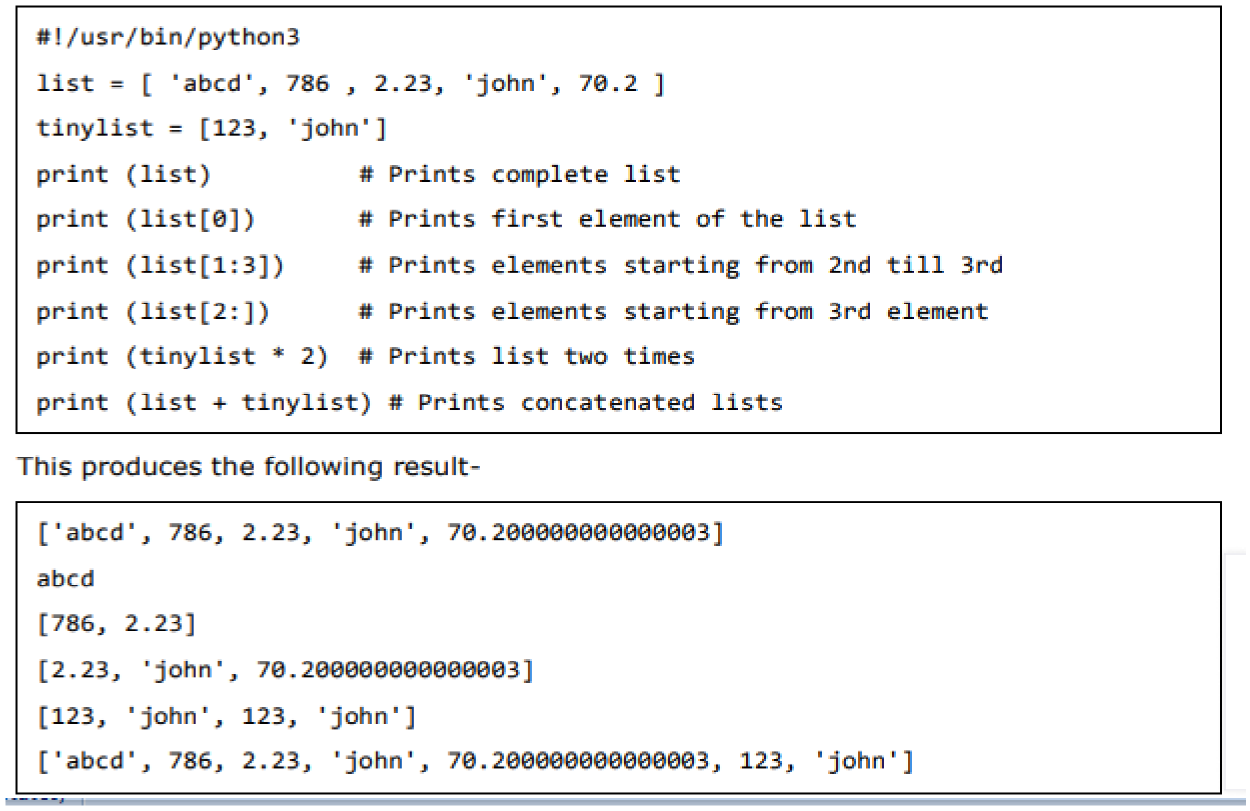
You must use j or J as imaginary component. Using other character will throw syntax error.

```
>>> a=5+2k
```

```
SyntaxError: invalid syntax
>>> a=5+j
SyntaxError: invalid syntax
>>> a=5j+2j
SyntaxError: invalid syntax
```

Python Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([ ]). To some extent, lists are similar to arrays in C. One of the differences between them is that all the items belonging to a list can be of different data type. The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator. For example-



```
#!/usr/bin/python3
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']
print (list)          # Prints complete list
print (list[0])       # Prints first element of the list
print (list[1:3])     # Prints elements starting from 2nd till 3rd
print (list[2:])      # Prints elements starting from 3rd element
print (tinylist * 2)  # Prints list two times
print (list + tinylist) # Prints concatenated lists
```

This produces the following result-

```
['abcd', 786, 2.23, 'john', 70.200000000000003]
abcd
[786, 2.23]
[2.23, 'john', 70.200000000000003]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john']
```

To delete an entire list, use the del keyword with the name of the list.

Python Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parenthesis. The main difference between lists and tuples is- Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as read-only lists.

i.e difference between the two (tuple and list) is that a list is mutable, but a tuple is immutable.

For example-

```
#!/usr/bin/python3
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
tinytuple = (123, 'john')
print (tuple)           # Prints complete tuple
print (tuple[0])        # Prints first element of the tuple
print (tuple[1:3])      # Prints elements starting from 2nd till 3rd
print (tuple[2:])       # Prints elements starting from 3rd element
print (tinytuple * 2)   # Prints tuple two times
print (tuple + tinytuple) # Prints concatenated tuple
```

This produces the following result-

```
('abcd', 786, 2.23, 'john', 70.200000000000003)
abcd
(786, 2.23)
(2.23, 'john', 70.200000000000003)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john')
```

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists

```
#!/usr/bin/python3
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
list = [ 'abcd', 786 , 2.23, 'john', 70.2  ]
tuple[2] = 1000    # Invalid syntax with tuple
list[2] = 1000     # Valid syntax with list
```

Python Set

A Python set is a slightly different concept from a list or a tuple.

Sets are a collection of unordered elements that are unique.

It does not hold duplicate values and is unordered. However, it is not immutable, unlike a tuple. Meaning that even if the data is repeated more than one time, it would be entered into the set only once.
Sets are created using the flower braces

```python
my_set = {1, 2, 3, 4, 5, 5, 5} #create set
print(my_set)
```

Output:

```
{1, 2, 3, 4, 5}
```

Python Dictionaries

Dictionaries are used to store data values in key:value pairs.

A dictionary is a collection which is ordered*, changeable and does not allow duplicates.

As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.

Dictionaries are written with curly brackets, and have keys and values:

```python
my_dict = {} #empty dictionary
print(my_dict)
my_dict = {1: 'Python', 2: 'Java'} #dictionary with elements
print(my_dict)
```

Output:

```
{}
{1: 'Python', 2: 'Java'}
```

Key Points to remember

Lists

- *List is a collection which is ordered.*
- *Lists are mutable (changeable) .*

- *Allows duplicate members*
- *Brackets used to represent: [ ]*
- *Lists are like arrays declared in other languages.*

Tuples

- *Collection of items which is ordered.*
- *Tuples are immutable (unchangeable) .*
- *Brackets used to represent: ( )*
- *Only difference between tuples and lists are that lists can be changed.*
- *Tuples are faster than lists as they are immutable.*

Sets

- *Collection of Unordered and Unindexed items.*
- *Sets are mutable (changeable).*
- *Does not take duplicate Values.*
- *Sets are unordered, so you cannot be sure in which order the items will appear.*
- *Brackets used to represent: { }.*
- *Sets are not faster than lists however they have a upper hand when it comes to membership testing.*

Dictionaries

- *Key:Value Pair in Python*
- *A dictionary is a collection which is ordered, changeable and indexed.*
- *In Python dictionaries are written with curly brackets, and they have keys and values.*
- *Brackets used to represent: { }.*

<u>Simple Definitions of list, tuple, set and dictionary:-</u>

List is mutable ordered collection of elements.

Tuple is immutable ordered collection of elements.

Set is mutable unordered unindexed collections of unique elements.

Dictionary is mutable ordered collection of key:value pairs.

-------**--------**

Topic 1.1.3- Operators

Operators

An operator is a symbol which performs operation on given data.

Python language supports the following types of operators-

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Arithmetic Operators The following are arithmetic operators supported by python with example.

Assume variable **a** holds the value 10 and variable **b** holds the value 21, then

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 31
- Subtraction	Subtracts right hand operand from left handoperand.	a – b = -11
* Multiplication	Multiplies values on either side of the operator	a * b = 210
/ Division	Divides left hand operand by right hand operand	b / a = 2.1
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 1
** Exponent	Performs exponential (power) calculation onoperators	a**b =10 to the power 20

//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed.	9//2 = 4 and 9.0//2.0 = 4.0

Example Program

```
a=10

b=21

print(a+b)

print(a-b)

print(a*b)

print(a/b)

print(a//b)

print(a%b)
```

Output

```
31
-11
210
0.47619047619047616
0
10
```

Relational or Comparision Operators

Comparision operators are used to compare values. It returns either True or False according to the condition.

Example Program

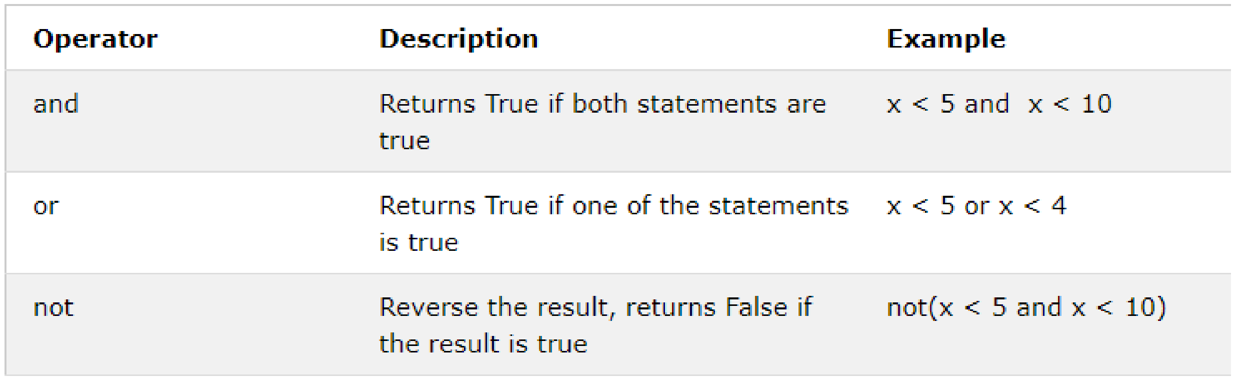
```
a=10
b=21
c=a>b
d=a<b
e=a>=b
f=a==b
g=a!=b
h=a<=b
print(c,d,e,f,g,h)
```

Output

```
False True False False True True
```

Logical Operators

These operators are used to compare two expressions. There are 3 logical operators which are mentioned below with example.



Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Example Program:-

```
a=21

b=10

c=a>b and a!=b

d=a>b or a==b

e=not(a!=b)

print(c,d,e)
```

Output:-

```
True True False
```

Identity Operators

Python provides two operators, is and is not, that determine whether the given operands have the same identity—that is, refer to the same object (same memory location). Two variables that are equal does not imply that they are identical.

Here is an example of two object that are equal but not identical:

Example 1:-

Python

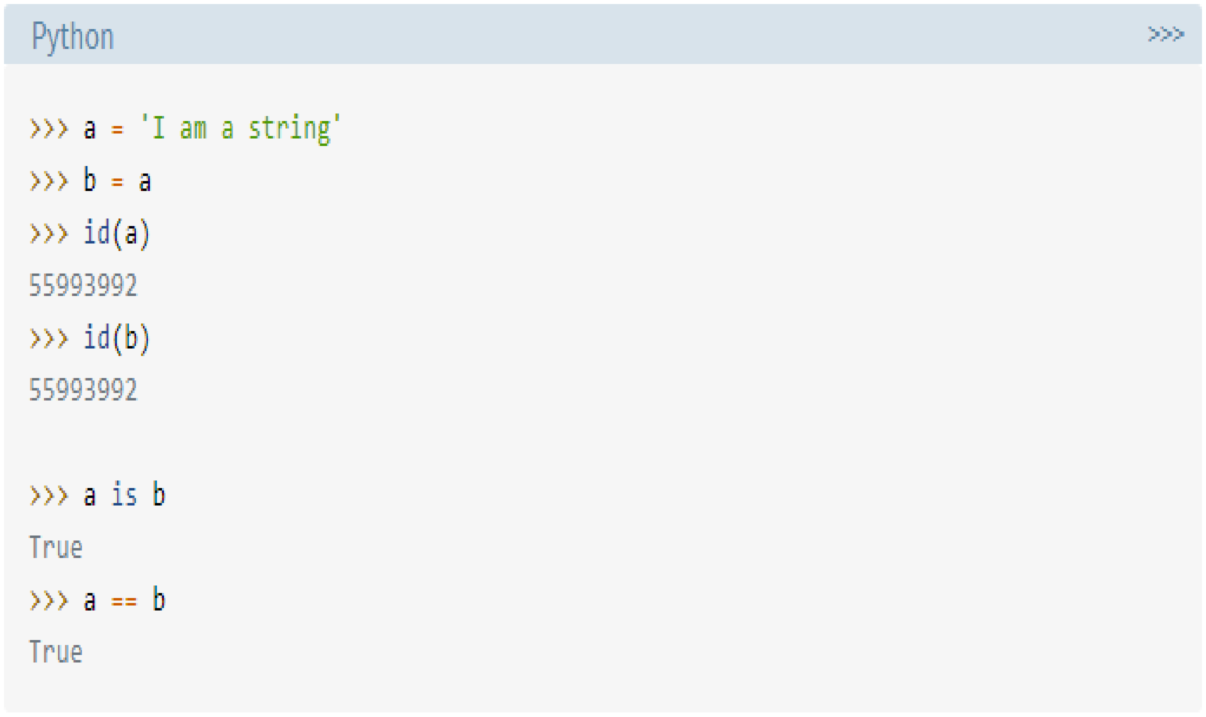
```
>>> x = 1001
>>> y = 1000 + 1
>>> print(x, y)
1001 1001

>>> x == y
True
>>> x is y
False
```

Here, x and y both refer to objects whose value is 1001. They are equal. But they do not reference the same object, as we can verify it using predefined function id( ).

Python

```
>>> id(x)
60307920
>>> id(y)
60307936
```

Example 2:-



```
>>> a = 'I am a string'
>>> b = a
>>> id(a)
55993992
>>> id(b)
55993992

>>> a is b
True
>>> a == b
True
```

Membership operators

in and not in are the membership operators in Python. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).

In a dictionary we can only test for presence of key, not the value.

Example 1:-

x = ["apple", "banana"]

print("banana" in x)

Output:-

True

Example 2:-

x = ["apple", "banana"]

print("pineapple" not in x)

Output:-

True

Bitwise Operators

Bitwise operators are used to perform bitwise calculations on integers.i,e Bitwise operator works on bits and performs bit-by-bit operation.
The integers are first converted into binary and then operations are performed on bit by bit, hence the name bitwise operators. Then the result is returned in decimal format.

Note: Python bitwise operators work only on integers.

Operator	**Operator Name**	**Example**
&	Bitwise AND	a & b
|	Bitwise OR	a | b
^	Bitwise XOR (exclusive OR)	a ^ b
~	Bitwise NOT	~a
<<	Bitwise left shift	a << n
>>	Bitwise right shift	a >> n

Bitwise AND operator: Returns 1 if both the bits are 1 else 0.
Example:
a = 10 = 1010 (Binary)

b = 4 = 0100 (Binary

a & b = 1010

&

0100

= 0000

= 0 (Decimal)

Bitwise or operator: Returns 1 if either of the bit is 1 else 0.
Example:

a = 10 = 1010 (Binary)

b = 4 = 0100 (Binary

a | b = 1010

| 0100

= 1110

= 14 (Decimal)

a = 10 = 1010 (Binary)

b = 4 = 0100 (Binary

Bitwisexor operator: Returns 1 if one of the bit is 1 and other is 0 else returns false.
Example:

```
a & b = 1010
        ^
       0100
     = 1110
     = 14 (Decimal)
```

Bitwise not operator: Returns one’s complement of the number.
Example:

```
a = 10 = 1010 (Binary)
~a = ~1010
   = -(1010 + 1)
   = -(1011)
   = -11 (Decimal)
```

Note:- Python’s built-in function bin( ) can be used to obtain binary representation of an integer number.
Ex:-

```
>>> bin(60)
'0b111100'
>>> bin(13)
'0b1101'
```

Left Shift in Python
The << (*Bitwise left shift*) operator, as its name suggests, shifts the bits towards the left to a number represented to the right side of this operator.
For example, 1 << 2 will shift 1 towards left for 2 values. In bit terms, it will be presented as follows:

- 1 = 0001.
- 1 << 2: 0001 << 2 = 0100 i.e. 4.

More examples:

- 14 << 1 = 01110 << 1 = 11100 = 28
- 24 << 4 = 000011000 << 4 = 110000000 = 384
- 19 << 3 = 00010011 << 3 = 10011000 = 152

Right Shift in Python

The >> (*right-shift*) operator, as its name suggests, shift the bits towards the right to a number represented to the right side of the operator.
For example, 10 >> 2 will shift the bits (1010) towards the right by 2.

- *10 = 1010*
- 10 >> 2*: 1010 >> 2 = 0010 = 2*

More examples:

- 14 >> 1 = 01110 >> 1 = 00111 = 7
- 24 >> 4 = 000011000 >> 4 = 00001 = 1
- 19 >> 3 = 00010011 >> 3 = 000010 = 2

<u>Example Program</u>

```
a=10

b=4

print('binar value of a = ',bin(a))

print('binar value of a = ',bin(b))

print('bitwise AND result = ',a&b)

print('bitwise OR result = ',a|b)

print('bitwise XOR result = ',a^b)

print('bitwise NOT result = ',~a)

a=24

print('binar value of a = ',bin(a))

c=a<<4

d=a>>4

print('left shift result = ',c)

print('right shift result = ',d)
```

<u>Output</u>

```
binar value of a =  0b1010

binar value of a =  0b100

bitwise AND result =  0

bitwise OR result =  14

bitwise XOR result =  14
bitwise NOT result =  -11
binar value of a =  0b11000
left shift result =  384
right shift result =  1
```

Assignment Operators

Assignment operators are used to assign values or result of an expression to variables.

a = 5 is a simple assignment operator that assigns the value 5 on the right to the variable a on the left.

There are various compound operators or short-hand assignment operatots in Python like a += 5 that adds to the variable and later assigns the same. It is equivalent to a = a + 5.

Operator	Example	Equivalent to
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5
^=	x ^= 5	x = x ^ 5

>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

Operator precedence and Associativity in python

An expression is a combination of operands, operators and constants (values).

To evaluate complex expressions, Python lays out the rule of precedence and associativity.

Precedence defines the order in which the operations take place in an expression.

And whenever two or more operators have the same precedence, then associativity defines the order of operations. Associativity can be either from left to right or right to left.

Below is the table containing the precedence and associativity of all operators in Python. The **operators are listed down in the decreasing order of**

Operator	**Name**	**Associativity**
( )	Parenthesis	Left to Right
**	Exponent	Right to left
~	Bitwise NOT	Left to right
*, /, %, //	Multiplication, Division, Modulo,	Left to right
+, –	Addition, Subtraction	Left to right
>>, <<	Bitwise right and left shift	Left to right
&	Bitwise AND	Left to right
^	Bitwise XOR	Left to right
|	Bitwise OR	Left to right
==, !=, >, <, >=, <=	Comparison	Left to right
=, +=, -=, *=, /=, %=, **=,	Assignment	Right to left
is, is not	Identity	Left to right

In, not in	Membership	Left to right
and, or, not	Logical	Left to right

Example 1:

```
num1, num2, num3 = 2, 3, 4
print ((num1 + num2) * num3)
```

Output:20

Example 2:

```
num1, num2, num3 = 2, 3, 4
print (num1 ** num2 + num3)
```

Output:12

Example 3:

```
num1, num2, num3 = 2, 3, 2
num4 = num1 ** num2 ** num3
print(num4)
```

Output:512

<u>Example Program</u>

```
a = 20
b = 10
c = 15
d = 5
e1 = (a + b) * c / d
print("Value of e1 is ",e1)
e2 = ((a + b) * c) / d
print("Value of e2 is ",e2)
e3 = (a + b) * (c / d);
print("Value of e3 is ",e3)
e4= a + b * c / d;
print("Value of e4 is ",e4)
```

<u>Output</u>

```
Value of e1 is  90.0
Value of e2 is  90.0
Value of e3 is  90.0
Value of e4 is  50.0
```

Type Conversion

The process of converting the value of one data type (integer, string, float, etc.) to another data type is called type conversion. Python has two types of type conversion.

1. Implicit Type Conversion
2. Explicit Type Conversion

Implicit Type Conversion or Automatic conversion

If two operands are of different data types, then lower data type is automatically converted to higher data type. The result is of higher data type.

In Implicit type conversion, Python automatically converts one data type to another data type. This process doesn't need any user involvement.

Let's see an example where Python promotes the conversion of the lower data type (integer) to the higher data type (float) to avoid data loss.

Example Program

```
#adding integer and float values
a=12
b=4.6
c=a+b
print(type(a),type(b),type(c))
print(c)
```

Output

```
<class 'int'> <class 'float'> <class 'float'>
16.6
```

Example program

```
# adding string and integer values
x=123
y='456'
z=x+y
print(type(x),type(y),type(z))
print(z)
```

Output

```
Traceback (most recent call last):
  File "J:/GAYATRI/Gayatri/PP&DS/python programs/implicit.py", line 10, in <module>
    z=x+y
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Explicit Type Conversion

In Explicit Type Conversion, users convert the data type of an object to required data type.

We use the predefined functions like int( ), float( ), str( ), etc to perform explicit type conversion.

This type of conversion is also called typecasting because the user casts (changes) the data type of the objects.

Syntax : <required_datatype>(expression)

Typecasting can be done by assigning the required data type function to the expression.

Example Program

```
a=123
b='456'
c=a+ int(b)     # In this example b is explicitly converted to int
print(type(a),type(b),type(c))
print('value of c is=',c)
```

Output

```
<class 'int'> <class 'str'> <class 'int'>
value of c is= 579
```

----**-----**

<u>Topic 1.1.4 – Input and Output</u>

Reading Input From the Keyboard

Programs often need to obtain data from the user, usually by way of input from the keyboard. The simplest way to accomplish this in Python is with input( ).

Syntax:- input([<prompt>])

Reads a line of input from the keyboard.

input( ) pauses program execution to allow the user to type in a line of input from the keyboard. Once the user presses the Enter key, all characters typed are read and returned as a string:

Example:-

```
>>> s=input( )
hello how are you
>>> s
'hello how are you'
```

If you include the optional <prompt> argument, input() displays it as a prompt to the user before pausing to read input:

Example:-

```
>>> name=input("what is your name?")
what is your name? Geethanjali
>>> name
' Geethanjali'
```

input( ) always returns a string. If you want a numeric type, then you need to convert the string to the appropriate type with the int( ), float( ), or complex( ) built-in functions:

Example:-

```
>>> n=input('enter n value')
enter n value7
>>> n
'7'
```

```
>>> type(n)

<class 'str'>

>>> n=n+10

Traceback (most recent call last):

  File "<pyshell#7>", line 1, in <module>

    n=n+10

TypeError: can only concatenate str (not "int") to str

>>> n=int(input('enter n value'))

enter n value7

>>> type(n)

<class 'int'>

>>> n=n+10

>>> n

17
```

Writing Output to the Console
In addition to obtaining data from the user, a program will also usually need to present data back to the user. You can display program data to the console in Python with print().

Unformatted Console Output

To display objects to the console, pass them as a comma-separated list of arguments to print( ).

Syntax:- print(<obj>, ..., <obj>)

Displays a string representation of each <obj> to the console.
By default, print( ) separates each object by a single space and appends a newline to the end of the output:

Example:-

```
>>> fname='geethanjali'

>>> lname='college'

>>> print(fname,lname)

geethanjali college
```

Any type of object can be specified as an argument to print( ). If an object isn’t a string, then print( ) converts it to an appropriate string representation displaying it:

Example:-

```
>>> a=[1,2,3]

>>> type(a)

<class 'list'>

>>> type(a)

<class 'list'>

>>> b={1:'abc',2:'xyz'}

>>> type(b)

<class 'dict'>

>>> c=(4,77.8)

>>> type(c)

<class 'tuple'>
```

Keyword Arguments to print( )

print( ) takes a few additional arguments that provide modest control over the format of the output. Each of these is a special type of argument called a **keyword argument**.

1) *The sep= Keyword Argument*

Adding the keyword argument sep=<str> causes objects to be separated by the

string <str> instead of the default single space:

Example without argument sep

```
>>> print('geethanjali','college')

geethanjali college
```

Example with argument sep

```
>>> print('geethanjali','college',sep='-')

geethanjali-college

>>> print('geethanjali','college',sep='----')
```

```
geethanjali----college
```

2) *The end= Keyword Argument*

The keyword argument end=<str> causes output to be terminated by <str> instead of the default newline:

Example program without end argument

```
print('pycharm')
print('spyder')
print('jupyter')
print('google colab')
print('idle')
```

Output

```
pycharm
spyder
jupyter
google colab
idle
```

Example program with end argument

```
print('pycharm',end='/')
print('spyder',end='/')
print('jupyter',end='/')
print('google colab',end='/')
print('idle')
```

Output

```
pycharm/spyder/jupyter/google colab/idle
```

Example :-

```
>>> for i in range(10):
        print(i)

0
1
2
3
4
5
6
7
8
9
>>> for i in range(10):
    print(i, end='  ')
0 1 2 3 4 5 6 7 8 9
```

The String Modulo Operator
The modulo operator (%) is usually used with numbers, in which case it computes remainder from division:

```
>>> 11 % 3
2
```

With string operands, the modulo operator has an entirely different function: string formatting.

Syntax:- `<format_string> % <values>`

On the left side of the % operator, <format_string> is a string containing one or more conversion specifiers. The <values> on the right side get inserted into <format_string> in place of the conversion specifiers.

Example:-

```
>>> print('%d %f %s hello' % (6,3.4,'welcome'))

6 3.400000 welcome hello

>>> print('%d %f hello %s $%.2f' % (3,5.6,'hello',1.7456))

3 5.600000 hello hello $1.75
```

Conversion Specifiers

Conversion specifiers appear in the <format_string> and determine how values are formatted when they're inserted.

A conversion specifier begins with a % character and consists of these components:

%[<flags>][<width>][.<precision>]<type>
% and <type> are required. The remaining components shown in square brackets are optional.

The following table summarizes what each component of a conversion specifier does:

Component	Meaning
%	Introduces the conversion specifier

Component	Meaning
<flags>	Indicates one or more flags that exert finer control over formatting
<width>	Specifies the minimum width of the formatted result
.<precision>	Determines the length and precision of floating point or string output
<type>	Indicates the type of conversion to be performed

Examples:-

```
>>> '%6d' % 1234, '%d' % 1234, '%2d' % 1234
('  1234', '1234', '1234')
>>> '%.2f' % 123.456789
'123.46'
>>> '%.2e' % 123.456789
'1.23e+02'
>>> '%.4s' % 'geethanjali'
'geet'
>>> '%4s' % 'geethanjali'
'geethanjali'
>>> '%6s' % 'hai'
'   hai'
>>> '%8.2f' % 123.45678
'  123.46'
>>> '%8.3s' % 'geethanjali'
'     gee'
```

------------*******-------------**

Topic 1.1.5 – Control Statements

A program’s **control flow** is the order in which the program’s code executes.

The control flow of a Python program is regulated by conditional statements, loops, and function calls.

Python has *three* types of control structures:

- **Sequential** - default mode
- **Selection** - used for decisions and branching
- **Repetition** - used for looping, i.e., repeating a piece of code multiple times.

1. Sequential

Sequential statements are a set of statements whose execution process happens in a sequence. The problem with sequential statements is that if the logic has broken in any one of the lines, then the complete source code execution will break.

Example :-

```
## This is a Sequential statement
a=20
b=10
c=a-b
print("Subtraction is : ",c)
```

Output:-

```
Subtraction is :  10
```

2. Selection/Decision control statements

In Python, the selection statements are also known as *Decision control statements* or *branching statements*.

The selection statement allows a program to test several conditions and execute instructions based on which condition is true.

Some Decision Control Statements are:

- Simple if
- if-else
- nested if

- if-elif-else

Simple if: *If statements* are control flow statements that help us to run a particular code, but only when a certain condition is met or satisfied. A *simple if* only has one condition to check.

Syntax:-

```
if test expression:

   statement(s)
```

Example Program :-

```
n = 10
if n % 2 == 0:
  print("n is an even number")
```

Output:-

```
n is an even number
```

if-else: The *if-else statement* evaluates the condition and will execute the body of if if the test condition is True, but if the condition is False, then the body of else is executed.

Syntax:-

```
if test expression:

   Body of if

else:

   Body of else
```

Example Program:-

```
# Program checks if the number is positive or negative
# And displays an appropriate message

num = 3
# Try these two variations as well.
# num = -5
# num = 0
if num >= 0:
   print("Positive or Zero")
else:
   print("Negative number")
```

Output:-

```
Positive or Zero
```

The elif Statement:

The **elif** statement allows you to check multiple expressions for TRUE and execute a blockof code as soon as one of the conditions evaluates to TRUE.

Syntax

```
if expression1:
     statement(s)
elif expression2:
     statement(s)
elif expression3:
     statement(s)
else:
     statement(s)
```

Example

```
amount=int(input("Enter amount: "))

if amount<1000:
        discount=amount*0.05
        print ("Discount",discount)
elif amount<5000:
        discount=amount*0.10
        print ("Discount",discount)
else:
        discount=amount*0.15
        print ("Discount",discount)
        print ("Net payable:",amount-
discount)
```

Input and Output:-

```
  Enter amount: 600
  Discount 30.0
  Net payable: 570.0

  Enter amount: 3000
  Discount 300.0
  Net payable: 2700.0

  Enter amount: 6000
  Discount 900.0
Net payable: 5100.0
```

Nested If statement:-

There may be a situation when you want to check for another condition after
a conditionresolves to true. In such a situation, you can use the nested **if**
construct.

In a nested **if** construct, you can have an **if...elif...else** construct inside another **if...elif...else** construct.

Syntax

```
if expression1:
    statement(s)
    if expression2:
        statement(s)
    elif expression3:
        statement(s)
    else
        statement(s)
elif expression4:
    statement(s)
else:
    statement(s)
```

Example:-

```
num=int(input("enter number"))
  if num%2==0:
      if num%3==0:
          print ("Divisible by 3 and 2")else:
          print ("divisible by 2 not divisible by 3")
  else:
      if num%3==0:
          print ("divisible by 3 not divisible by 2")else:
print   ("not Divisible by 2 not divisible by 3")
```

Input and Output:-

```
  enter number8
 divisible by 2 not divisible by 3

 enter number15
 divisible by 3 not divisible by 2

 enter number12
 Divisible by 3 and 2

 enter number5
not Divisible by 2 not divisible by 3
```

3. Repitition Statements:-

A loop statement allows us to execute a statement or group of statements multiple times.

Python has two loop statements:

- while loop
- for loop

While Loop statement:-

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.

We generally use this loop when we don't know the number of times to iterate beforehand.

Syntax:-

```
while test_expression:
        statement(s)
```

In the while loop, test expression is checked first. The body of the loop is entered only if the test_expression evaluates to True. After one iteration, the test expression is checked again. This process continues until the test_expression evaluates to False.

In Python, the body of the while loop is determined through indentation.

The body starts with indentation and the first unindented line marks the end.

```
# Program to add natural numbers up to n ie sum = 1+2+3+...+n

# To take input from the user,


n = int(input("Enter n: "))

# initialize sum and counter
sum = 0
i = 1

while i <= n:
   sum = sum + i
   i = i+1    # update counter

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

Input and Output:-

```
Enter n: 10

The sum is 55
```

for loop statement :-

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

Syntax:-

```
for iterating_var in sequence:
        statements(s)
```

Example Program:-

```
# Program to find the sum of all numbers stored in a list
# List of numbers
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]
# variable to store the sum
sum = 0
# iterate over the list
for val in numbers:
   sum = sum+val
print("The sum is", sum)
```

Output:-

```
The sum is 48
```

The range( ) function:-

We can generate a sequence of numbers using `range( )` function. `range(10)` will generate numbers from 0 to 9 (10 numbers).
We can also define the start, stop and step size as `range(start, stop,step_size)`. step_size defaults to 1 if not provided.

Example Program:-

```
print(range(10))
print(list(range(10)))
print(list(range(2, 8)))
print(list(range(2, 20, 3)))
```

Output:-

```
range(0, 10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[2, 3, 4, 5, 6, 7]
[2, 5, 8, 11, 14, 17]
```

We can use the `range( )` function in `for` loops to iterate through a sequence of numbers. It can be combined with the `len( )` function to iterate through a sequence using indexing

Example Program:-

```
# Program to iterate through a list using indexing
colleges= ['GIST', 'SVCN', 'NEC']
# iterate over the list using index
for i in range(len(colleges)):
   print("Engineering College is ", colleges[i])
```

Output:-

```
Engineering College is GIST
Engineering College is SVCN
 Engineering College is NEC
```

----**----**

UNIT-I
PART-II

TOPICS:-

1. Strings: Creating strings and basic operations on strings, string testing methods.
2. Lists
3. Dictionaries
4. Tuples

Topic 1.2.1 Strings

A string is a sequence of characters enclosed in quotation marks. In Python, strings can be created by enclosing the character or the sequence of characters in the quotes. Python allows us to use single quotes, double quotes, or triple quotes to create the string.

Note:- Python strings are "immutable" which means they cannot be changed after they are created .

Creating String in Python:-

We can create a string by enclosing the characters in single-quotes or double- quotes. Python also provides triple-quotes to represent the string, but it is generally used for multiline string or docstrings.

Example Program:-

```
#Using single quotes
str1 = 'Hello Python'
print(str1)

#Using double quotes
str2 = "Hello Python"
print(str2)

#Using triple quotes
str3 = '''Triple quotes are generally used for
   represent the multiline or
   docstring'''
print(str3)
```

Output:-

```
Hello Python
Hello Python
```

Triple quotes are generally used for
represent the multiline or
docstring
Like other languages, the indexing of the Python strings starts from 0.

Strings indexing and slicing:-

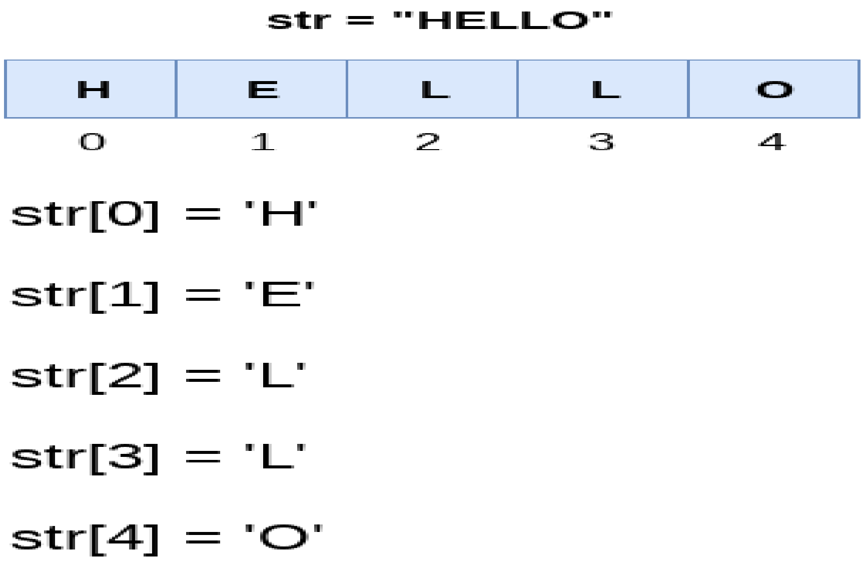
Indexing: Indexing is used to obtain individual elements.

Slicing: Slicing is used to obtain a sequence of elements.

String Indexing:-
To access characters in a string we have two ways:

- **Positive index number -** The index number starts from **0** which denotes the **first character** of a string.
- **Negative index number -** index number starts from index number **-1** which denotes the **last character** of a string.

For example, The string "HELLO" is indexed as given in the below figure.



Example Program 1:-

```
str = "HELLO"
print(str[0])
print(str[1])
print(str[2])
print(str[3])
print(str[4])
# It returns the IndexError because 6th index doesn't exist
print(str[6])
```

Output:-

```
H
E
L
L
O
IndexError: string index out of range
```

Example Program 2:-

```
my_str = "Python"
print(my_str[-1])
```

Output:-

```
P
```

String Slicing:-

Slicing in python is used for accessing parts of a sequence. The slice object is used to slice a given sequence or any object. We use slicing when we require a part of a string and not the complete string.

Syntax:

```
string[start : end : step]
```

Example program 1:-

```
# Given String
str = "GEETHANJALI"
# Start 0th index to end
print(str[0:])
# Starts 1th index to 4th index
print(str[1:5])
# Starts 2nd index to 3rd index
print(str[2:4])
# Starts 0th to 2nd index
print(str[:3])
#Starts 4th to 6th index
print(str[4:7])
```

Output:-

```
GEETHANJALI
EETH
ET
GEE
HAN
```

Example program 2:-

```
str = 'GEETHANJALI COLLEGE'
print(str[-1])
print(str[-3])
print(str[-2:])
print(str[-4:-1])
print(str[-7:-2])
# Reversing the given string
print(str[::-1])
print(str[-12])
```

Output:-

```
E
E
GE
LEG
COLLE
EGELLOC ILAJNAHTEEG
J
```

Python has a set of built-in methods that you can use on strings.
Note: All string methods returns new values. They do not change the original string.

Python String len( ) method

String len( ) method eturn the length of the string.

```
str = "Hellow World!"
print(len(str))
```

output:-
```
13
```

Python String count( ) method

String count( ) method returns the number of occurrences of a substring in the given string.

```
str = "Python is Object Oriented"
substr = "Object"
print(str.count(substr)) # return 1, becasue the word Object exist 1 time in str
```

Output:-

```
1
```

Python String index( ) method

String index( ) method returns the index of a substring inside the given string.

Syntax:-

index(substr,start,end)

```
str = "Python is Object Oriented"
    substr = "is"
    print(str.index(substr))
```

Output:-

```
7
```

Python String upper() method

String upper( ) convert the given string into Uppercase letters and return new string.

```
str = "Python is Object Oriented"
print(str.upper( ))
```

output:-

```
PYTHON IS OBJECT ORIENTED
```

Python String lower( ) method

String lower( ) convert the given string into Lowercase letters and return new string.

```
str = "Python is Object Oriented"
print(str.lower())

output:-
python is object oriented
```

Python String startswith( ) method

String startswith( ) method returns Boolean TRUE, if the string Starts with the specified substring otherwise, it will return False.

```
str = "Python is Object Oriented"
print(str.startswith("Python"))
print(str.startswith("Object"))

output:-
True
False
```

Python String endswith( ) method

String endswith( ) method returns Boolean TRUE, if the string Ends with the specified substring otherwise, it will return False.

```
str = "Python is Object Oriented"
print(str.endswith("Oriented"))
print(str.endswith("Object"))

output:-
True
False
```

Python String split( ) method

String split() method break up a string into smaller strings based on a delimiter or character.

```
str = 'Python is Object Oriented'
print(str.split( ))

output:-
['Python', 'is', 'Object', 'Oriented']


str = 'Python,is,Object,Oriented'
print(str.split(','))

output:-
['Python', 'is', 'Object', 'Oriented']
```

Python String find( ) method

String find( ) return the index position of the first occurrence of a specified string. It will return -1, if the specified string is not found.

```
str = "Python is Object Oriented"
st = "Object"
print (str.find(st))
print (str.find(st,20)) #finding from 20th position

output:-
10
-1
```

Python String strip( ) method

String strip( ) remove the specified characters from both Right hand side and Left hand side of a string (By default, White spaces) and returns the new string.

```
str = "   Python is Object Oriented   "
print (str.strip( ))

output:-
```

Python is Object Oriented

Python String rstrip( ) method

String rstrip() returns a copy of the string with trailing characters removed.

```
str = "   Python is Object Oriented   "
print (str.rstrip( ))
```

output:-
```
Python is Object Oriented
```

Python String lstrip( ) method

String lstrip( ) returns a copy of the string with leading characters removed.

```
str = "   Python is Object Oriented   "
print (str.lstrip( ))
```

output:-
```
Python is Object Oriented
```

Topic 1.2.2 Lists in python :-

A list in Python is used to store the sequence of various types of data. A list can be defined as a collection of values or items of different types. The items in the list are separated with the comma (,) and enclosed with the square brackets [ ].

Example:-

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5 ];
list3 = ["a", "b", "c", "d"]
```

List Items

List items are ordered, changeable(mutable), and allow duplicate values.

Ordered

When we say that lists are ordered, it means that the items have a defined order, and that order will not change.

If you add new items to a list, the new items will be placed at the end of the list.

Changeable

The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

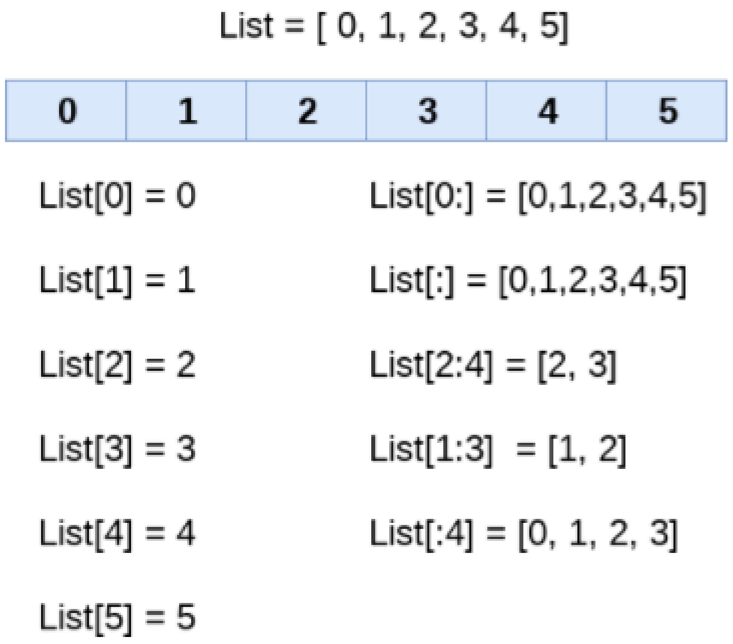
Allow Duplicates

Since lists are indexed, lists can have items with the same value

List indexing and slicing

The indexing is processed in the same way as it happens with the strings. The elements of the list can be accessed by using the slice operator [ ].

The index starts from 0 and goes to length - 1. The first element of the list is stored at the 0th index, the second element of the list is stored at the 1st index, and so on.



```
List = [ 0, 1, 2, 3, 4, 5]

List[0] = 0          List[0:] = [0,1,2,3,4,5]
List[1] = 1          List[:] = [0,1,2,3,4,5]
List[2] = 2          List[2:4] = [2, 3]
List[3] = 3          List[1:3]  = [1, 2]
List[4] = 4          List[:4] = [0, 1, 2, 3]
List[5] = 5
```

We can get the sub-list of the list using the following syntax.

list_varible(start:stop:step)

Example Program:-

```
list = [1,2,3,4,5,6,7]
print(list[0])
print(list[1])
print(list[2])
print(list[3])
# Slicing the elements
print(list[0:6])
# By default the index value is 0 so its starts from the 0th element and go for index -1.
print(list[:])
print(list[2:5])
print(list[1:6:2])
```

Output:-

```
1
2
3
4
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6, 7]
[3, 4, 5]
[2, 4, 6]
```

Slice Elements

Python slice extracts elements, based on a start and stop.

```
str = ['h','e','l','l','o']
```

```
lsc = str[1:4]
print(lsc)
```

Output:-

```
['e', 'l', 'l']
```

Example:-

```
str = ['h','e','l','l','o']
lsc = str[:3] # slice first three elements
print(lsc)
```

Output:-

```
['h', 'e', 'l']
```

Example:-

```
str = ['h','e','l','l','o']
lsc = str[3:] # slice from 4th element, Python starts its lists at 0 rather than 1.
print(lsc)
```

Output:-

```
['l', 'o']
```

List Methods:-

Method	**Description**
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list

count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

List length

The function len returns the length of a list, which is equal to the number of its elements.

```
numbers = ['one','two','three','four','five']

list_len = len(numbers)

print(list_len)
```

output:-

```
5
```

Clear or Emptying List

list.clear() remove all items from the list.

```
numbers = ['one','two','three','four','five']
print(numbers)
numbers.clear()
print(numbers)
```

output:-

```
['one', 'two', 'three', 'four', 'five']
[]
```

Inserting and Removing Elements

append() - Appends adds its argument as a single element to the end of a list. The length of the list itself will increase by one.

```
numbers = ['one','two','three','four','five']
numbers.append('six')
print(numbers)
```

output:-

```
['one', 'two', 'three', 'four', 'five', 'six']
```

Appending a list inside a list

```
num1 =[1,2,3]
num2 = [4,5,6]
num1.append(num2)
print(num1)
```

output:-

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

List operations

Using the "+" operator concatenates lists.

```
num1 =[1,2,3]
num2 = [4,5,6]
num3 = num1 + num2
print(num3)
```

output:-

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

using the * operator repeats a list a given number of times.

```
num1 =['hi']
num = num1 * 4
print(num)
```

output:-

```
['hi', 'hi', 'hi', 'hi']
```

```
num1 =[1,2,3,4]
num = num1 * 2
print(num)
```

output:-

```
[1, 2, 3, 4, 1, 2, 3, 4]
```

Inserting elements in List

Example 1:-

```
num1 =[1,2,4]
num1.insert(2,3) #inserts an element into the third position
print(num1)
```

output:-

```
[1, 2, 3, 4]
```

Example 2:-

```
num1.insert(-1,5) #inserts an element into the second from last position of the list
(negative indices start from the end of the list)
print(num1)
```

Output:-

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

Remove elements from List

```
numbers = ['one','two','three','four','five']
numbers.remove('three')
print(numbers)
```

Output:-

```
['one', 'two', 'four', 'five']
```

List Count

list.count(x) return the number of times x appears in the list.

```
str = ['h','e','l','l','o']
cnt = str.count('l')
print(cnt)
```

Output:-

```
2
```

List Reverse

The reverse() method in list reverse the elements of the list in place.

```
str = ['h','e','l','l','o']

str.reverse()

print(str)
```

Output:-

```
['o', 'l', 'l', 'e', 'h']
```

List index()

The index() method returned the index of the first matching item.

```
str = ['h','e','l','l','o']

ind = str.index('l') # from start 'l' is in 2nd position

print(ind)
```

Output:-

```
2
```

If you want to specify a range of valid index, you can indicate the start and stop indices:

example

```
str = ['h','e','l','l','o']

ind = str.index('l',3) # start searching from 3rd position

print(ind)
```

Output:-

```
3
```

Exist in List

We can test if an item exists in a list or not, using the keyword "in"

```
str = ['h','e','l','l','o']
```

```
print('e' in str)
```

Output:-

```
True
```

not in List

```
str = ['h','e','l','l','o']
print('e' not in str)
```

Output:-

```
False
```

List sort

List sort() method that performs an in-place sorting

```
str = ['h','e','l','l','o']
str.sort()
print(str)
```

Output:-

```
['e', 'h', 'l', 'l', 'o']
```

Reverse Sorting

example

```
str = ['h','e','l','l','o']
str.sort(reverse=True)
print(str)
```

Output:-

```
['o', 'l', 'l', 'h', 'e']
```

Reverse items

example
str = [1,2,3,4,5,6,7,8,9,10]

print(str[::-1])

Output:-

[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

Python list extend method()

The list **extend() method** extends the list by adding all items of a list (passed as an argument) to the end.

Syntax:-

list.extend(anotherList)

Example:-

list1 = ['a', 'b', 'c']

list1.extend(['d', 'e'])

print (list1)

output:-

['a', 'b', 'c', 'd', 'e']

Append() vs extend() in lists:-

Python **append() method** adds an element to a list, and the **extend() method** concatenates the first list with another list (or another iterable). When append() method adds its argument as a single element to the end of a list, the length of the list itself will increase by one. Whereas **extend() method** iterates over its argument adding each element to the list, extending the list. The length of the list will increase by however many elements were in the iterable argument.

Python append() example
list1 = ['a', 'b', 'c']

list1.append(['d', 'e'])

print (list1)

Output:-

['a', 'b', 'c', ['d', 'e']]

Python extends( ) example

```
list1 = ['a', 'b', 'c']

list1.extend(['d', 'e'])

print (list1)
```

Output:-

```
['a', 'b', 'c', 'd', 'e']
```

Topic 1.2.3 – Tuples in python

A **Tuple** is a collection of immutable **Python** objects separated by commas. Tuples are just like lists, but we cannot change the elements of a tuple once it is assigned whereas in a list, elements can be changed.

Creating a Tuple

A tuple is defined using parenthesis. An empty tuple can be formed by an empty pair of parentheses.

```
a_tuple = () #empty tuple

print(a_tuple)
```

Output:-

```
()
```

Creating Tuple with values

```
a_tuple = ('East','West','North','South')

print(a_tuple)
```

Output:-

```
('East', 'West', 'North', 'South')
```

Python Tuple with mixed datatypes

example

```
a_tuple = (1,2,'sunday','monday',3.14)

print(a_tuple)
```

Output:-

```
(1, 2, 'sunday', 'monday', 3.14)
```

Tuple Items

Tuple items are ordered, unchangeable(immutable), and allow duplicate values.

Unpacking a Tuple

When we create a tuple, we normally assign values to it. This is called "packing" a tuple:

Packing a tuple:

Example:-

```
fruits = ("apple", "banana", "cherry")

print(fruits)
```

Output:-

```
("apple", "banana", "cherry")
```

Unpacking a tuple:

Example:-

```
fruits = ("apple", "banana", "cherry")
(green, yellow, red) = fruits
print(green)
print(yellow)
print(red)
```

Output:-

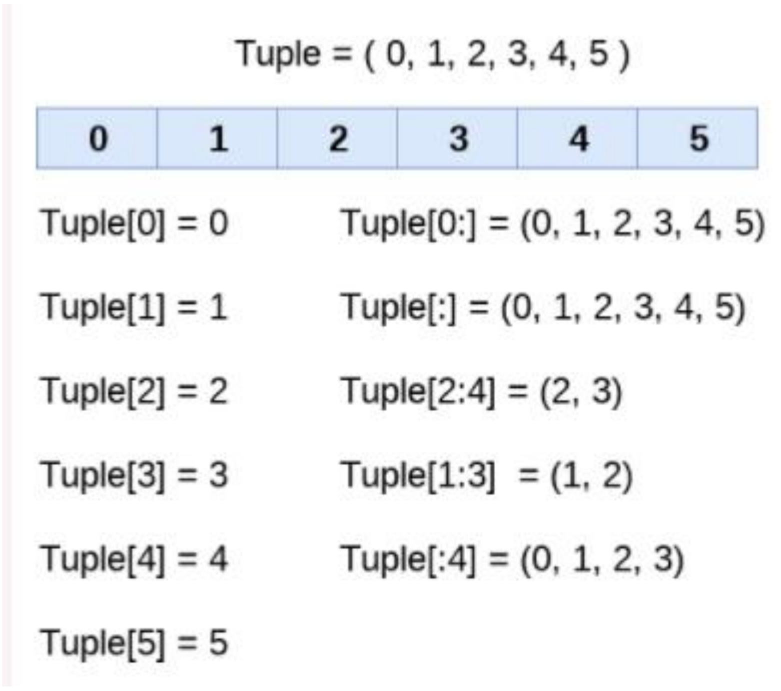
```
apple

banana

cherry
```

Tuple indexing and slicing

The indexing and slicing in the tuple are similar to lists. The indexing in the tuple starts from 0 and goes to length(tuple) - 1.

The items in the tuple can be accessed by using the index [] operator. Python also allows us to use the colon operator to access multiple items in the tuple.

Consider the following image to understand the indexing and slicing in detail.



```
Tuple[0] = 0        Tuple[0:] = (0, 1, 2, 3, 4, 5)
Tuple[1] = 1        Tuple[:] = (0, 1, 2, 3, 4, 5)
Tuple[2] = 2        Tuple[2:4] = (2, 3)
Tuple[3] = 3        Tuple[1:3]  = (1, 2)
Tuple[4] = 4        Tuple[:4] = (0, 1, 2, 3)
Tuple[5] = 5
```

Example Program:-

```
tuple = (1,2,3,4,5,6,7)

#element 1 to end

print(tuple[1:])

#element 0 to 3 element

print(tuple[:4])

#element 1 to 4 element

print(tuple[1:5])
```

```
# element 0 to 6 and take step of 2

print(tuple[0:6:2])
```

Output:-

```
(2, 3, 4, 5, 6, 7)

(1, 2, 3, 4)

(2, 3, 4, 5)

(1, 3, 5)
```

Example 2:-

```
tuple1 = (1, 2, 3, 4, 5)
print(tuple1[-1])
print(tuple1[-4])
print(tuple1[-3:-1])
print(tuple1[:-1])
print(tuple1[-2:])
```

Output:-

```
5

2

(3, 4)

(1, 2, 3, 4)

(4, 5)
```

Python has two built-in methods that you can use on tuples.

Method	Description
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

Example:-

```
my_tuple = ('a', 'p', 'p', 'l', 'e',)

print(my_tuple.count('p'))  # Output: 2

print(my_tuple.index('l'))  # Output: 3
```

Output:-

```
2

3
```

Deleting Tuple

Unlike lists, the tuple items cannot be deleted by using the **del** keyword as tuples are immutable. To delete an entire tuple, we can use the **del** keyword with the tuple name.

Other Tuple Operations

Tuple Membership Test

We can test if an item exists in a tuple or not, using the keyword in.

```
# Membership test in tuple

my_tuple = ('a', 'p', 'p', 'l', 'e',)

# In operation

print('a' in my_tuple)
print('b' in my_tuple)

# Not in operation

print('g' not in my_tuple)
```

```
Output:-

True

False

True
```

Topic 1.2.4 - Dictionary in python

Dictionaries are used to store data values in key:value pairs.

A dictionary is a collection which is ordered*, changeable and does not allow duplicates.

Creating the dictionary

The dictionary can be created by using multiple key-value pairs enclosed with the curly brackets {}, and each key is separated from its value by the colon (:).

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
print(type(Employee))
print("printing Employee data .... ")
print(Employee)
```

```
Output:-
<class 'dict'>
printing Employee data ....
{'Name': 'John', 'Age': 29, 'salary': 25000, 'Company': 'GOOGLE'}
```

Accessing the dictionary values

The values can be accessed in the dictionary by using the keys as keys are unique in the dictionary.

Example:-

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
print(type(Employee))
print("printing Employee data .... ")
print("Name : %s" %Employee["Name"])
print("Age : %d" %Employee["Age"])
print("Salary : %d" %Employee["salary"])
print("Company : %s" %Employee["Company"])
```

Output:-

```
<class 'dict'>
printing Employee data ....
Name : John
Age : 29
Salary : 25000
Company : GOOGLE
```

Updating Dictionary contents

```
students = {}
students['ID']=1001
students['Name']="John"
students['Age']=22
print (students)
students['Age'] = 23 # update age
students['Country'] = 'USA' # add new entry
print (students)
```

Output:-

```
{'Name': 'John', 'ID': 1001, 'Age': 22}
{'Name': 'John', 'ID': 1001, 'Country': 'USA', 'Age': 23}
```

Retrieve whole Dictionary entries

We can use loop through whole entries in a Dictionary. When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the items() method.

Example:-

```
directions = {}
directions[1]="East"
directions[2]="West"
directions[3]="North"
directions[4]="South"
for key, value in directions.items():
  print(key, value)
```

Output:-

```
1 East

2 West

3 North

4 South
```

Retrieve Keys and Values in a List

Example:-

```
directions = {}

directions[1]="East"

directions[2]="West"

directions[3]="North"

directions[4]="South"

keys = list(directions.keys())

print(keys)

vals = list(directions.values())

print(vals)
```

Output:-

```
[1, 2, 3, 4]

['East', 'West', 'North', 'South']
```

Python has a set of built-in methods that you can use on dictionaries.

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs

values()	Returns a list of all the values in the dictionary

Copying a Dictionary

A dictionary can be copied with the method copy()

Example:-

```
directions = {}
directions[1]="East"
directions[2]="West"
directions[3]="North"
directions[4]="South"
dir1 = directions.copy()
print(dir1)
```

Output:-

```
{1: 'East', 2: 'West', 3: 'North', 4: 'South'}
```

Merging Dictionaries

The update() method of Dictionary is to merges the keys and values of one dictionary into another, overwriting values of the same key.

```
dict1 = {1:'a',2:'b',3:'c'}
dict2 = {4:'d',5:'e'}
dict1.update(dict2)
print(dict1)
```

Output:-

```
{1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e'}
```

How to sort a dictionary?

The operator module also defines tools for generalized attribute and item lookups. These are useful for making fast field extractors as arguments for map(), sorted(), itertools.groupby(), or other functions that expect a function argument.

How to sort a dictionary by key?

```
import operator
x = {1: 2, 3: 4, 4: 3, 2: 1, 0: 0}
```

```
sorted_x = sorted(x.items(), key=operator.itemgetter(0))
print(sorted_x)
```

Output:-

```
[(0, 0), (1, 2), (2, 1), (3, 4), (4, 3)]
```

How to sort a dictionary by value?

```
import operator
x = {1: 2, 3: 4, 4: 3, 2: 1, 0: 0}
sorted_x = sorted(x.items(), key=operator.itemgetter(1))
print(sorted_x)
```

Output:-

```
[(0, 0), (2, 1), (1, 2), (4, 3), (3, 4)]
```

Remove values from Dictionaries

You can remove an entry from Dictionaries using the key.

Example:-

```
directions = {}
directions[1]="East"
directions[2]="West"
directions[3]="North"
directions[4]="South"
del directions[2]
print (directions)
```

Output:-

```
{1: 'East', 3: 'North', 4: 'South'}
```

The clear() method of Dictionary remove all entries in dictionary

Example:-

```
directions = {}
directions[1]="East"
directions[2]="West"
directions[3]="North"
directions[4]="South"
directions.clear()
print (directions)
```

```
Output:-
{}
```

If you want to delete entire dictionary, you can use "del" command.

Example:-

```
directions = {}
directions[1]="East"
directions[2]="West"
directions[3]="North"
directions[4]="South"
del directions
print (directions)
```

```
Output:- (errror will be raised)
Traceback (most recent call last):
  File "sample.py", line 11, in & module &
    print (directions)
NameError: name 'directions' is not defined
```

Other Dictionary Operations

```
# Membership Test for Dictionary Keys

squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

print(1 in squares)

print(2 not in squares)

# membership tests for key only not value

print(49 in squares)
```

```
Output:-
True
True
False
```

Pop( ):-

The **pop()** method accepts the key as an argument and remove the associated value.

Example:-

```
# Creating a Dictionary
Dict = {1: 'JavaTpoint', 2: 'Peter', 3: 'Thomas'}
# Deleting a key
# using pop() method
pop_ele = Dict.pop(3)
print(Dict)
```

Output:-

```
{1: 'JavaTpoint', 2: 'Peter'}
```

Topic :- Sets in python

Python **Set** is like a dictionary, an unordered collection of keys, held without any values. The set type is **mutable** , the contents can be changed using methods like **add()** and **remove()** . Sets do not contain duplicates, every element in a Python set can have only one occurrence in that particular set, no multiple occurrences are allowed.

Example:-

```
set_num = set("Hello World!")
print(set_num)
```

Output:-

```
{'o', 'H', 'r', 'l', ' ', 'd', 'e', '!', 'W'}
```

Sets are Mutable

The set type is mutable, the contents can be changed using methods like add( ) and

remove( ).

Example:-

```
set_col = set({'Red','Blue','Green'})
print(set_col)
set_col.add('White')
print(set_col)
```

Output:-

```
{'Green', 'Blue', 'Red'}
{'White', 'Green', 'Blue', 'Red'}
```

Set.remove( ) remove elements from set.

Example:-

```
set_col = set({'Red','Blue','Green'})
print(set_col)
set_col.remove('Blue')
print(set_col)
```

Output:-
```
{'Blue', 'Green', 'Red'}
{'Green', 'Red'}
```

Set Length

len( ) function returned the length of a set.

```
set_col = set({'Red','Blue','Green'})
print(len(set_col))
```

Output:-
```
3
```

Union of Sets

In Python, Union operation can be performed on two Python sets using the operator or by using the method union(). Union of two sets x and y is another set that contains all distinct elements in sets x and x.

Using | Operator

example
```
x = set({1,2,3,4,5})
y = set({3,4,5,6,7})
z = x  y
print(z)
```

Output:-
```
 {1, 2, 3, 4, 5, 6, 7}
```

Using union( )

example
```
x = set({1,2,3,4,5})
y = set({3,4,5,6,7})
z = x.union(y)
```

```
print(z)
```

Output:-
```
{1, 2, 3, 4, 5, 6, 7}
```

Intersection of Sets

In Python, Union operation can be performed on two Python sets using the operator & or by using the method intersection(). Intersection of two sets x and y is another set of elements those are present in both sets x and y (common to both of them).

Using & operator

example
```
x = set({1,2,3,4,5})
y = set({3,4,5,6,7})
z = x & y
print(z)
```

Output:-
```
{3, 4, 5}
```

Using intersection( )

example

```
x = set({1,2,3,4,5})
y = set({3,4,5,6,7})
z = x.intersection(y)
print(z)
```

Output:-
```
{3, 4, 5}
```

Difference of Sets

The difference() method returns the set difference of two sets. In Python, Difference operation can be performed on two Python sets using the operator - or by using the method difference().

Using - operator

example
```
x = set({1,2,3,4,5})
```

```
y = set({3,4,5,6,7})
z = x-y
print(z)
```

Output:-
```
{1, 2}
```

Using difference( )

example
```
x = set({1,2,3,4,5})
y = set({3,4,5,6,7})
z = x.difference(y)
print(z)
```

Output:-
```
{1, 2}
```

Symmetric Difference of Sets

Symmetric difference of the sets is the set of elements those are not common to both the sets. The symmetric_difference( ) method using.

Using symmetric_difference( )

example
```
x = set({1,2,3,4,5})
y = set({3,4,5,6,7})
z = x.symmetric_difference(y)
print(z)
```

Output:-
```
{1, 2, 6, 7}
```

----**** THE END****----