3. Language Fundamentals:

To prepare Python applications, we need some constructs bydefault from Python Programming language called as "Language Fundamentals".

To prepare Python applications, Python has provided the following fundamentals.
1. Tokens
2. Data Types
3. Type Casting
4. Python Statements

1. Tokens:

Lexeme: Logical Individual Unit in programming is called as Lexeme.
Token: The Collection of lexemes come under a particular group called as Token.

EX:

```
a = b + c * d

Lexemes: a, =, b, +, c, *, d  ---> 7 Lexemes

Tokens:
Identifiers: a, b, c, d
Operators: =, +, *

Tokens: 2 types of tokens
```

To prepare Python applications, Python has provided the following tokens.
1. Identifiers
2. Literals
3. Keywords/Reserved Words
4. Operators

1. Identifiers:

Providing names to the programming elements like variables, functions, classes, ...
called as "Identifiers".

To provide identifiers in Python applications we have to follow a set of rules and regulations.

1. All python identifiers must not be started with a number, they may start with an alphabet or _ symbol, but, the subsequent symbols may be a number, an alphabet or _ symbol.

EX:

```
eno = 111 ---> Valid
9eno = 9999 ---> Invalid
emp9Name = "Durga" ---> Valid
_ename = "Durga" ---> Valid
emp_Addr = "Hyd" --> Valid
$esal = 5000.0 ---> Invalid
```

2. All Python identifiers are not allowing all operators and all

special symbols except _ symbol.

```
EX:
---
empName = "Durga" ---> valid
emp.Name = "Durga" ---> Invalid
emp+Addr = "Hyd" ---> Invalid
emp-Sal = 5000.0 ---> Invalid
emp_Addr = "Hyd" ---> Valid
emp@Hyd = "Durga"  ---> Invalid
```

3. All Python Identifiers are not allowing spaces in the middle.

```
EX:
---
empName = "Durga" ---> Valid
emp Name = "Durga" ---> Invalid
tempEmpAddr = "Hyd" ---> Valid
temp Emp Addr = "Hyd" ---> INvalid
```

4. In Python applications, we are unable to use keywords and reserved words as identifiers.

```
EX:
if = 10 --> INvalid
for = 20 ---> Invalid
```

5. In Python applications, we are able to use all fundamental data types names and sequence data types names as identifiers.

```
EX:
int = 10 --> Valid
float = 20 ---> Valid
bool = 50 ---> Valid
list = 100 --> Valid
set = 200 --> Valid
tuple = 70 --> Valid
list = [10, 20, 30, 40, 50] --> Valid
```

6. All Python identifiers are case sensitive.

```
EX:
---
ename = "Durga"
ENAME = "Ravi"
print(ename)
print(ENAME)
OP:
---
Durga
Ravi
```

Along with the above rules and regulations, Python has provided the following suggestions to provide identifiers.

1.In Python applications, it is suggestible to provide identifiers with a particular meaning.

```
EX:
xxx = "abc123" ----> Not Suggestible
accNo = "abc123"  ----> Suggestible
```

2. In Python applications, there is no length restriction for the identifiers , we can prepare identifiers with any length but it is

```
suggestible to manage length of the identifiers around 10 symbols.
EX:
temporaryemployeeaddress = "Hyd" ---> Not Suggestible
tempEmpAddr = "Hyd" ---> Suggestible

3. If we have multiple words in an identifiers then it is suggestible
to seperate multiple words with the special notations like '_' symbols.
EX:
tempEmpAddr = "Hyd" ---> Not Suggestible
temp_Emp_Addr = "Hyd" --> Suggestible

Q)What is the difference between variable and identifier?
------------------------------------------------------------
Ans:
----
Variable is a memory location where data is stored, where name of the
variable is called as an Identifier.

Note: In Programmnig Languages, when we access variable then we are
able to get the data which was stored at the respective memory
location, we are unable to get memory location directly.

2. Literals:
------------
Literal is a constant assigned to the variable
EX:
a = 10
a ---> variable/Identifier
= ---> Operator
10 --> Constant[Literal]

There are two types of literals in Python.
1. Numeric Literals
2. Non Numeric Literals

1. Numeric Literals:
---------------------
1. int literals : 10, 20, 30,....
2. float Literals : 123.234, 456.345,...

Note: Upto Python2.x version 'long' data type is existed and its
literals are existed, but, from Python3.x version long data type is not
existed and its literals are not existed.

EX:
a = 10
print(a)
print(type(a))
OP:
10
<class 'int'>

EX:
a = 234.345
print(a)
print(type(a))
OP:
234.345
<class 'float'>
```

```
2. Non Numeric Literals:
------------------------
1. bool: True, False
2. str : if we provide any thing in ' '[Single quatotions], " "[Double
quatotions],
         ''' '''[Triple Quatotions] then that literal is str literal.

Note: To represent string data in a single line then we are able to use
'' or "" or ''' ''' , but, to represent data in multiple lines then we
are able to use ''' ''' only.

EX:
---
#b = true --> Error
b = True
print(b)
print(type(b))

c = False
print(c)
print(type(c))
OP:
True
<class 'bool'>
False
<class 'bool'>

EX:
---
s1 = 'Durga Software Solutions'
print(s1)
print(type(s1))
OP:
---
Durga Software Solutions
<class 'str'>

EX:
---
s2 = "Durga Software Solutions"
print(s2)
print(type(s2))
OP:
Durga Software Solutions
<class 'str'>

EX:
---
s3 = '''Durga Software Solutions'''
print(s3)
print(type(s3))
OP:
Durga Software Solutions
<class 'str'>

EX:
---
str = '''Durga
str = '''Durga
Software
Solutions'''
print(str)
print(type(str))
OP:
Durga
Software
Solutions
<class 'str'>
```

In String literals, it is not possible to provide single quatotions insise single quatotions directly, but, it is possible with \, that is , \'.

```
EX:
---
str = 'Durga 'Software' Solutions'
Status: Syntax Error.

EX:
---
str = 'Durga \'Software\' Solutions'
print(str)
OP:
Durga 'Software' Solutions
```

In String literals, we are able to provide double quotations inside single quotation directly.

```
EX:
---
str = 'Durga "Software" Solutions'
print(str)
OP:
Durga "Software" Solutions
```

In String literals, it is not possible to provide triple quotations inside single quotation with or with /.

```
EX:
---
str = 'Durga '''Software''' Solutions'
Status: Syntax Error

EX:
---
str = 'Durga \'''Software\''' Solutions'
OP:
Durga 'Software' Solutions
```

In String literals, in side double quotations, we are able to provide single quotations and triple quotations directly ,but, we are unable to provide double quotations directly, but, we are able to provide doubles quotations with \.

```
EX:
---
str1 = "Durga 'Software' Solutions"
print(str1)
#str2 = "Durga "Software" Solutions" --> Error
str2 = "Durga \"Software\" Solutions"
print(str2)
```

```
str3 = "Durga '''Software''' Solutions"
print(str3)
```

OP:

```
Durga 'Software' Solutions
Durga "Software" Solutions
Durga '''Software''' Solutions
```

In String literals, inside triple quotations, we are able to provide single quotations and double quotations directly, we are unable to provide triple quotations directly , but, it is possible to provide triple quotations with \.
EX:

```
str1 = '''Durga 'Software' Solutions'''
print(str1)
str2 = '''Durga "Software" Solutions'''
print(str2)
#str3 = '''Durga '''Software''' Solutions''' --> Error
str3 = '''Durga \'''Software\''' Solutions'''
print(str3)
```
OP:

```
Durga 'Software' Solutions
Durga "Software" Solutions
Durga '''Software''' Solutions
```

```
Q)Find the right options to display the following string literal
   'Durga' "Software" '''Solutions'''
1)str = '\'Durga\' "Software" \'''Solutions\''''
  print(str)
2)str = "'Durga' \"Software\" '''Solutions'''"
  print(str)
3)str = ''''Durga' "Software" \'''Solutions\''''''
  print(str)
4)None
```

Ans: 2, 3

```
Q) eno = 111
   ename = 'Durga'
   esal = 50000.0
```
Find data types order of the variables ename,eno and esal respectovily?
1. str, float, int
2. str, int, float
3. float, str, int
4. None

Ans: 2

Number Systems:

In all the programming languiages, to represent numbers we have to use some standards, they are Number Systems.

There are four number systems in Programming Languages
1. Binary Number System[BASE-2]
2. Octal Number System[BASE-8]

```
2. Octal Number System[BASE-8]
3. Decimal Number System[BASE-10]
4. Hexa Decimal Number System[BASE-16]

All the four number systems are allowed in python , but, the default
number system is "Decimal Number System".

1. Binary Number System[BASE-2]
---------------------------------
Alphbet: 0, 1
Prefix : 0b or 0B
EX:
---
a = 0b1010 ---> Valid
b = 0B1100 ---> Valid
c = 0b1020 ---> Invalid
d = 01100 ----> Invalid
e = 10 -------> It is not binary number, it is decimal number.

2. Octal Number System:
-----------------------
Alphabets: 0,1,2,3,4,5,6 and 7.
Prefix   : 0o or 0O [zero and O ]
EX:
---
a = 10   ------> It is not octal number, it is decimal number
b = 01234 -----> Invalid
c = 0o4567-----> Valid
d = 0O34567----> Valid
e = 0O5678-----> Invalid
f = 0023456----> Invalid

3. Decimal Number System:
--------------------------
Alphabet : 0,1,2,3,4,5,6,7,8,9
Prefix: No prefix
EX:
a = 10 ---------> valid
b = 123456 -----> Valid
c = 56789 ------> Valid
d = 789abc -----> Invalid
e = 012345 -----> Invalid

4. Hexa Decimal Number System:
-------------------------------
Alphabet : 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f
Prefix : 0x or 0X
EX:
a = 10  ----------> It is decimal number, it is not hexa decimal number
b = 0x1234567 ----> Valid
c = 0X6789abc ----> valid
d = 0x123def -----> Valid
e = 0Xadefg ------> Invalid
```

Q)Match the following

a. Binary Number System	1. 1234
b. Octal number System	2. 0o23456
c. Decimal number System	3. 0X123abc
d. Hexa Decimal Number System	4. 0B1010

d. hexa Decimal Number System   4. 0B1010

a)a-1, b-2, c-3, d-4
b)a-4, b-2, c-1, d-3
c)a-1, c-2, d-3,c-4
d)a-3, b-1, c-4, d-2

Ans: b

Note: In Python applications, if we provide any number in any number system then PVM will recognize the provided number and its number system on the basis of prefix value , PVM will convert that provided number from the specified number system to decimal number System, PVM will process that converted number as like decimal number and PVM will display that numbner as like Decimal number.

In Python applications, we are able to perform arithmetic operations over the numbers which are existed in different number systems
EX:

```
a = 0b1010 // 10
b = 0B1100 // 12
c = a + b
print("a :", a)
print("b :", b)
print("c :", c)
```
OP:
```
a : 10
b : 12
c : 22
```

EX:

```
a = 0b1010
b = 0O10
c = a + b
d = a - b
e = a * b
f = a / b
print("a :", a)
print("b :", b)
print("c :", c)
print("d :", d)
print("e :", e)
print("f :", f)
```
OP
```
a: 10
b: 8
c: 18
d: 2
e: 80
f: 1.25
```

In Python applications, we are able to convert numbers from one number system to another system by using the following predefined functions.

1. bin()
2. oct()
3. hex()

1. bin()

This predefined function can be used to convert all numbers from all number systems into Binary Number System
EX:

```
a = 10
b = 0o10
c = 0x10
dec_To_Binary = bin(a)
oct_To_Binary = bin(b)
hex_To_Binary = bin(c)
print(dec_To_Binary)
print(oct_To_Binary)
print(hex_To_Binary)
```
OP:

```
0b1010
0b1000
0b10000
```

2. oct()

This predefined function can be used to convert all numbers from all the number systems to Octal number system.
EX:

```
a = 0b1010
b = 10
c = 0x10
binary_To_Oct = oct(a)
dec_To_Oct = oct(b)
hex_To_oct = oct(c)
print(binary_To_Oct)
print(dec_To_Oct)
print(hex_To_oct
```
OP:

```
0o12
0o12
0o20
```

3. hex()

This predefined function can be used to convert all numbers from all number systems to Hexa Decimal Number System.
EX:

```
a = 0b1010
b = 10
c = 0o10
binary_To_Hex = hex(a)
dec_To_Hex = hex(b)
oct_To_Hex = hex(c)
print(binary_To_Hex)
print(dec_To_Hex)
print(oct_To_Hex)
```
OP:

OP:

```
0xa
0xa
0x8
```

Note: In Python , the predefined functions like bin(), oct() and hex() will return a value of type 'str'.
EX:

```
print(type(bin(10)))
print(type(oct(10)))
print(type(hex(10)))
```
OP:

```
<class 'str'>
<class 'str'>
<class 'str'>
```

Note: In Python, type() predefined function can be used to return type of the variable or data.

3. Keywords/Reserved Words

If any predefined word has both word recognization and internal functionality then that predefined word is called as Keyword or Reserved Word.

To prepare python applications, Python has provided the following set of Keywords.

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

All the above keywords are provided by Python software in the form of keyword.py file at "C:\Python\Python37\Lib".

To get all Python Keywords on console we have to use the following code.
```
import keyword
print(keyword.kwlist)
```

EX:

```
C:\Python\Python37\Lib\keyword.py
-----------------------------------
kwlist=['False', 'None', 'True', 'and',.....]
```

Note: In Python, all keywords are provided in lower case letters except True, None and False. In True, False and None only first symbol is Upper case letter.

Q)Find the valid keyword in Python?
a)true
b)none
c)False

d)None of the above
Ans: C

4. Operators:

Operator is a symbol, it will perform a particular operation over the provided operands.

To prepare Python applications, Python has provided the following list of Operators.

1. Arithmetic Operators

+, -, *, **[Power operator], /, //[Floor Division], %[Modulo]

2. Assignment Operators:

=, +=, -=, *=, /=, %=

3. Comparision Operators:

==, !=, <, >, <=, >=

4. Logical Boolean operators:

&, |, ^, not

Note: In Python, we can rerpesent & and | operators in word form also like 'and' and 'or'.

5. Logical Bitwise Operators:

&[and], |[or], ^, <<, >>, ~

Note: ~ is called as Bitwise 1's compliment Operator.

6. Ternary Operator:

Expr1 if Expr2 else Expr3

7. Identity Operators:

is, is not

8. Membership Operators:

in, not in

Example on Arithmetic Operators:

```
a = 10
b = 5
print("a :", a)
print("b :", b)
print("a+b :", a+b)
print("a-b :", a-b)
print("a*b :", a*b)
print("a/b :", a/b)
```

```
[illegible]

OP:
a : 10
b : 5
a+b : 15
a-b : 5
a*b : 50
a/b : 2.0

Q)What is the difference between * and ** operators?
------------------------------------------------------
Ans:
----
In Python applications, * operator will be used to perform Arithmetic
Multiplication operation.

In Python applications, ** operator is representing power operator.
EX:
---
a = 10
b = 3
print("a :", a)
print("b :", b)
print("a*b :",a*b)
print("a**b :",a**b)

OP:
---
a : 10
b : 3
a*b : 30
a**b : 1000

Q)What is the difference between / and // operators?
------------------------------------------------------
Ans:
----
In Python applications, / operator will perform Arithmetic division
operation and it will return always float value as result whether the
operands are int or float.

In Python applications, // operator will perform Arithmetic Division
operator and it will return the results as per the following
conditions.
1. If both the operands are int type then the result of // operator is
int only.
2. If either of the operands or both the operands are float then the
result of //      operator is float.
EX:
---
a = 11
b = 2
c = a / b
print(c)
print(type(c))
OP:
---
5.5
[illegible]
```

```
<class 'float'>

EX:
a = 5.5
b = 2.2
c = a / b
print(c)
print(type(c))
OP:
----
2.5
<class 'float'>

EX:
---
a = 10
b = 2
c = a // b
print(c)
print(type(c))
OP:
---
5
<class 'int'>

EX:
----
a = 5.5
b = 2
c = a // b
print(c)
print(type(c))
OP:
----
2.0
<class 'float'>

EX:
----
a = 5
b = 2.2
c = a // b
print(c)
print(type(c))
OP:
----
2.0
<class 'float'>

EX:
---
a = 5.5
b = 2.2
c = a // b
print(c)
print(type(c))
OP:
----
2.0
```

```
<class 'float'>
```

Example on Assignment Operators:

```
a = 10
print(a)
a += 2
print(a)
a -= 5
print(a)
a *= 2
print(a)
a /= 2
print(a)
a %= 2
print(a)
```

OP:

```
10
12
7
14
7.0
1.0
```

Example on Comparision Operator:

```
a = 10
b = 20
print(a == b)
print(a != b)
print(a < b)
print(a <= b)
print(a > b)
print(a >= b)
```

OP:

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

Example on Logical Boolean Operators:

If we want to evaluate boolean operators then we have to use the following table.

A	B	A&B	A|b	A^B	not A
T	T	T	T	F	F
T	F	F	T	T	F
F	T	F	T	T	T
F	F	F	F	F	T

```
a = True
```

```
a = True
b = False
print(a&a)
print(a&b)
print(b&a)
print(b&b)
print(a and b)
print()
print(a|a)
print(a|b)
print(b|a)
print(b|b)
print(a or b)
print()
print(a^a)
print(a^b)
print(b^a)
print(b^b)
print()
print(not a)
print(not b)
```

OP:

```
True
False
False
False
False

True
True
True
False
True

False
True
True
False

False
True
```

Example on Logical Bitwise Operators:

To evaluate Logical bitwise operators we have to use the following table.

A	B	A&B	A|B	A^B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

```
a = 10
b = 2
print(a & b)
print(a | b)
print(a ^ b)
print(~a)
print(a << b)
print(a >> b)

OP:
---
2
10
8
-11
40
2

Evaluation:
a = 10 ----> 1010
b = 2  ----> 0010
-------------------
   a&b ----> 0010 -----> 2
   a|b ----> 1010 -----> 10
   a^b ----> 1000 -----> 8
   ~a -----> ~10 -------> -10-1 = -11
   a<<b ---> 10 << 2 --> 00001010
                           00101000 ---> 40
   a>>b ---> 10 >> 2 --> 00001010
                       00000010 -------> 2
```

Example on Ternary Operator:

Syntax: Expr1 if Expr2 else Expr3
EX:

```
a = 10
b = 20

min = a if a < b else b
max = a if a > b else b

print("a :", a)
print("b :", b)
print("min :", min)
print("max :", max)

OP:
a : 10
b : 20
min : 10
max : 20
```

Example on Identity Operators:

The main intention of Identity operators is to check whether the two variables values are same or not same.

In Python, there are two Identity operators.
1. is
2. is not

Where 'is' operator will check whether the provided two variables values are same or not, if the variables values are same then 'is' operator will return True value , if the two variables values are not same then 'is' operator will return false value.

Where 'is not' operator will check whether the provided two variables values are not same or not, if the provided two variables values are not same then 'is not' operator will return True value, if the provided two variables values are same then 'is not' operator will return False value.

```
EX:
a = 10
b = 10
c = 20
print(a is b)
print(a is c)
print(a is not b)
print(a is not c)

OP:
---
True
False
False
True
```

Example on Membership Operators:

The main intention of membership operators is to check whether the specified element is the members of the specified sequence type or not.

There are two types member ship operators in Python.
1. in
2. not in

Where 'in' operator will check whether the specified element is member of the specified sequence type or not, if the specified element is member of the specified sequence then 'in' operator will return True value , if the specified element is not the member of the specified sequence then 'in' operator will return False value.

Where 'not in' operator will check whether the specified element is not the member of the specified sequence or not, if the specified element is not the member in the specified sequence then 'not in' operator will return True value, if the specified element is existed in the specified sequence then 'not in' operator will return False value.

```
EX:
---
l = [1,2,3,4,5]
print(1 in l)
print(10 in l)
print(5 not in l)
print(10 not in l)
OP:
---
True
False
```

```
False
True

EX:
---
Consider the following Python code.
a = 10
b = 'abc'
c = 22.22
d = True
e = +23456E456

Provide the data types of the variables a,b,c,d respectivily.
Ans: int, str, float, bool, float

EX:
Consider the following Code:
a = 10
b = 2
print(a/b)
print(a//b)
print(a%b)

What will be the result.
Ans: 5.0, 5, 0

EX:
Consider the following Code
a = 13
b = 3
print(a/b)
print(a//b)
print(a%b)
What will be the Result.
Ans: 4.3, 4, 1

EX:
Consider the following Code
a = 12.5
b = 4
print(a/b)
print(a//b)
print(a%b)
What is the result.
Ans:3.1, 3.0, 0.5

EX:
---
a = 5
b = 2
print(a*b)
print(a**b)
What is the result.
Ans: 10, 25

EX:
----
a = 5
print(a+2*a+1)
```

```
Ans:16

EX:
---
a = 13
print(a/3//2)
Ans: 2.0
EX:
---
a = 6
print(a*2**2)
Ans:24

a*2**2
6*2**2
6*4
24

Operators Priorities order Decreasing Mode:
id
()
Exponents, **
*, /,//, %
+, -
and

EX:
a = 5+6*4/2
print(a)
Ans: 17.0

EX:
a = 2 * (3 + 4 ** 2) + 3 * (4 ** 2 - 2)
print(a)
Ans:  80

2 * (3 + 4 ** 2) + 3 * (4 ** 2 - 2)
2 * (3 + 16) + 3 * (4 ** 2 - 2)
2 * (3 + 16) + 3 * (16 - 2)
2 * 19 + 3 * (16 - 2)
2 * 19 + 3 * 14
38 + 3 * 14
38 + 42
80


EX:
---
a = 8 // 6 % 5 + 2 ** 3 - 2
print(a)
Ans: 7

8 // 6 % 5 + 2 ** 3 - 2

1 % 5 + 2 ** 3 - 2

1 % 5 + 8 - 2

1 + 8 - 2
```

```
9 - 2
7
```

EX:

```
a = 3
b = 10
a += 3**2
a -= b // 2 // 3
print(a)
```

Ans:11

```
a = 3
b = 10
a += 3**2 --> a = a + 3 ** 2 --> a = 3 + 9 --> a = 12
a -= b // 2 // 3 --> a = a - b // 2 // 3 --> a = 12 - 10 // 2 // 3
                                           a = 12 - 5 // 3
                                           a = 12 - 1
                                           a = 11
```

EX:
Which of the following expression will generate max value
1. 8%3*4
2. 8-3*4
3. 8//3*4
4. 8/3*4

Ans: 4

```
print(8%3*4)--> 8
print(8-3*4) --> -4
print(8//3*4)---> 8
print(8/3*4) ----> 10.666666
```

/ and // are having same priority.

2. Data Types:

In any programming language if the data is provided as per the classifications or types then that programming language is called as Typed Programming language.

There are two types of programming Languages as per the Data types.
1. Statically Typed Programming language.
2. Dynamically Typed Programming Language.

1. Statically Typed Programming language:

In any Proigramming Language, if we are providing data types before representing data then that programming language is called as Statically Typed Programming Language.
EX:

```
1. int a = 10; ---> valid
2. int a;
   a = 20; -------> Valid
3. a = 30; -------> Invalid
```

Note: C, C++ and Java are statically Typed Programming Languages.

2. Dynamically Typed Programming Language:

In any programming language, if we represent data with out providing data types explicitly and if the data types are calculated after providing data then that programming languages are called as Dynamically Typed Programming Languages.
EX: Python.

```
EX: a = 10 ---> valid
EX: int a = 20; --> Invalid.
```

Note: In Python , every data is represented in the form of an object and these objects are come under the respective data types.

In Python, it is possible to get data type of the provided data by using a predefined function like type(--)
EX:
```
a = 10
print(a)
print(type(a))
```
OP:
```
<class 'int'>
```

To prepare Python applications, Python has provided the following data types.

```
1. Fundamental Data Types
   a)Numeric Data Types
      int
      long
      float
      complex
      Note: Upto Python2.x version long data is existed, but, from
Python3.x                    verison long data type is not existed.
   b)Non Numeric Data Types
      None
      bool
      str
2. Sequence Data Types / Advanced Data Types
   1. bytes
   2. bytearray
   3. range
   4. list
   5. tuple
   6. set
   7. frozenset
   8. dict
```

int :

--> It able to store only integral data.
--> Its objects are immutable objects, these objects are not allowing modifications on their content, if we are trying to perform modifications over the data then data is allowed for modification but the resultent modified data will be stored by creating new object.
EX:
```
a = 10
```

```
print(a)
print(type(a))
print(id(a))
OP:
10
<class 'int'>
140726104257056
```

EX:
```
a = 10
print(a)
print(id(a))
a = a + 10
print(a)
print(id(a))
```

OP:
```
10
140726104257056
20
140726104257376
```

In Pythn applications, we are able to represent numbers in binary, octal, decimal and hexa decimal as int type data.
EX:

```
a = 0b1010
b = 0o123
c = 10
d = 0x123abc
print(type(a))
print(type(b))
print(type(c))
print(type(d))
OP:
<class 'int'>
<class 'int'>
<class 'int'>
<class 'int'>
```

Q)How many no of objects are created for int type in the following Python code?
```
a = 10
b = 10
c = 10
print(a)
print(b)
print(c)
```

Ans: 1

```
print(id(a))
print(id(b))
print(id(c))
OP:
140726104257056
140726104257056
140726104257056
```

```
EX:
a = 10
a = a + 10
a = a + 10
print(a)
OP: 30

Ans: 3 objects are created.

Explanation:
--------------
a = 10
print(id(a))
a = a + 10
print(id(a))
a = a + 10
print(id(a))
print(a)

OP:
---
140725848994336
140725848994656
140725848994976
30

2. float:
----------
--> It includes integral part and fractional part.
--> Its objects are immutable objects.
EX:
---
f = 12.234
print(f)
print(type(f))
print(id(f))
OP:
---
12.234
<class 'float'>
1910119359920
EX:
---
f = 12.22
print(id(f))
f = f + 10
print(id(f))
f = f + 10
print(id(f))
print(f)
OP:
3222635177392
3222635177328
3222635177360
32.22

3. Complex Type:
------------------
```

--> Its data includes real value and imaginary value.
--> To represent data in Compex Type we have to use the following syntax.

```
     a+bj
```

Where a is real part
Where b is imaginary part

EX:

```
a = 10+2j
print(a)
print(type(a))
print(id(a))
```

OP
```
(10+2j)
<class 'complex'>
1863307100496
```

In Complex Numbers, we are able to get real and imaginary data seperatly by using the predefined variables "real" and "imag".
EX:

```
a = 10+2j
print(a)
print(a.real)
print(a.imag)
```

OP:
```
(10+2j)
10.0
2.0
```

In Complex Numbers, we are able to provide int, float values as real part and amaginary part, but, bydefault, they will take float values.
EX:

```
a = 22.22+2.5j
print(a)
print(type(a))
```

OP:
```
(22.22+2.5j)
<class 'complex'>
```

In Complex Numbers, we are able to provide real value by using the number systems like binary number system, octal number system , decimal number system and hexa decimal number system, but, imaginary part is able to allow the value in decimal number system only.

Note: In Complex numbers if we provide real and imaginary values in any number system then PVM will convert these numbers from the specified number system to decimal number system of float type and PVM will process that numbers as like float values.
EX:

```
a = 0o10 + 2j
print(a)
print(type(a))
print(a.real)
print(a.imag)
```

In Python applications, we are able to perform all the arithmetic operations like addition, subtraction , Multiplication,.. over the complex numbers.

EX:
```
---
a = 10 + 10j
b = 5 + 5j
c = a + b
d = a - b
e = a * b
print(a)
print(b)
print(c)
print(d)
print(e)
```
OP:
```
---
(10+10j)
(5+5j)
(15+15j)
(5+5j)
100j
```
Expl:
```
----
a = 10 + 10j
b = 5 + 5j

a+b --> (10+5) + (10+5)j --> 15 + 15j
a-b --> (10-5) + (10-5)j --> 5 + 5j

a*b --> 10*5 + 10*5j + 10j*5 + 10j*5j
                            2
    --> 50 + 50j + 50j + 50j
    --> 50 + 100j - 50
    --> 100j
```

In Python, Complex data type objects are immutable objects.

EX:
```
---
a = 10 + 10j
print(a)
print(id(a))
a = a + (5 + 5j)
print(a)
print(id(a))
```
OP:
```
---
(10+10j)
2510474314160
(15+15j)
2510474314320
```

None Type:

In Python applications, it is not possible to declare any variable with

```
out explicit initialization, we must provide intialization for the
variables explicitly.

a --> Invalid
a = 10 --> Valid

In python applicartions, if we want to declare a variable with no data
then we have to use 'None' as value to the variables, where 'None' is
representing no data.
EX:
---
a = None
print(a)
print(type(a))
print(id(a))

OP:
---
None
<class 'NoneType'>
140726009281760

bool:
-----
This data type is able to provide two values like True and False .
EX:
---
b = True
print(b)
print(type(b))
print(id(b))
OP:
---
True
<class 'bool'>
140726009235792

In Python, bool type objects are immutable objects.
EX:
---
b = False
print(b)
print(id(b))
b = True
print(b)
print(id(b))
OP:
---
False
140726009235824
True
140726009235792

In python , boolean values True and False are represented internally in
the form of 1 and 0 respectivly and it is possible to perform Arthmetic
Operations like Addition, Subtraction, Multiplication,... over boolean
values.
EX:
---
```

```
a = True
print(int(a))
b = False
print(int(b))
c = True + False
print(c)
d = True - False
print(d)
f = True * False
print(f)
OP:
---
1
0
1
1
0

str:
----
It is the collection of alphabets, digits, special symbols,........
IN Python applications, we are able to represent str values by using '
or  " or
''' .
EX:
---
a = 'Durga Software Solutions'
b = "Durga Software Solutions"
c = '''Durga
Software
Solutions'''
print(a)
print(b)
print(c)
OP:
---
Durga Software Solutions
Durga Software Solutions
Durga
Software
Solutions

str type objects are immutable objects.
EX:
---
a = "Durga "
b = a + "Software "
c = b + "Solutions"
print(a)
print(b)
print(c)
print(id(a))
print(id(b))
print(id(c))

OP:
---
Durga
Durga Software
Durga Software
Durga Software Solutions
1693420140400
1693421470960
1693420046096
```

In Python applications, we are able to read elements from String individually on the basis of index values. In str values, we are able to read elements in both forward direction and backward direction. In forward direction we must use +ve index values and it must be started from 0 and in backward direction we must use
-ve index values and it must be started with -1.

To retrive elements from str type on the basis of index values we have to use the following syntax.

```
         refVar[index_Value]
```

Note: If we provide index_Value in outside range of the String index values then PVM will provide an error like "IndexError: string index out of range".

EX:

```
str = "Durga"
print(str)
print(str[2])
print(str[4])
print(str[-2])
print(str[-4])
#print(str[5]) --> IndexError
print(str[-5])
#print(str[-6]) --> IndexError
```
OP:

```
Durga
r
a
g
u
D
```

Note: To get length of the string we have to use len(--) predefined function.
EX:

```
str = "Durga Software Solutions"
print(str)
print(len(str))
```
OP:
```
Durga Software Solutions
24
```

In Python applications, if we want to read elements from String type we will use for loop.
EX:

```
str = "Durga Software Solutions"
for x in range(0, len(str)):
    print(x,"--->",str[x])
print()
```

```
for x in range(0, len(str)):
    index = -(x+1)
    print(index,"--->",str[index])

OP:
---
0 --> D
1 --> u
-------
-------
24 --> s

-1 --> s
-2 --> n
-----
-----
-25 --> D

EX:
---
str = "Durga Software Solutions"
for x in str:
    print(x)

OP:
D
u
--
--
s
```

bytes Type:

-->It is a sequence data type, it able to represent sequence of numbers.
-->It able to allow the numbers from 0 to 255.
-->To represent numbers in bytes data we have to use the following steps.
      1. Represent numbers in List by using [].
      2. Convert numbers from List to bytes data type by using bytes() function.

```
EX:
---
list = [1,2,3,4,5]
print(list)
print(type(list))
b = bytes(list)
print(b)
print(type(b))

OP:
---
[1, 2, 3, 4, 5]
<class 'list'>
b'\x01\x02\x03\x04\x05'
<class 'bytes'>
```

In Python applications, bytes data type is able to allow the numbers from 0 to 255, if we provide numbers in out side range of 0 to 255 then

PVM will provide an error like "ValueError: bytes must be in the range (0, 256)"

```
EX:
---
l = [253, 254, 255, 256]
b = bytes(l)
print(b)

Status: ValueError: bytes must be in range(0, 256)
```

bytes objects are immutable objects, they are not allowing modifications on their content, if we are trying to perform modifications then PVM will raise an error like "TypeError: 'bytes' object does not support item assignment"

```
EX:
---
l = [1,2,3,4,5]
b = bytes(l)
print(b)
b[2] = 10
print(b)

Status: "TypeError: 'bytes' object does not support item assignment"
```

In bytes type we are able to retrive numbers individually by providing index value, if the index value is in out side range of the bytes index values then PVM will rise an error like "indexError: index out of range".

In bytes type we are able to read elements in both forward direction and backward direction, to read elements in forward direction we have to provide +ve index values and it will be started with 0. If we want to read elements in backward direction then we have to provide -ve index values and it must be started with -1.

```
EX:
---
l = [1,2,3,4,5,6,7,8,9,10]
b = bytes(l)
print(b)
print(b[2])
print(b[-2])
print(b[6])
print(b[-5])
#print(b[10])-->IndexError: index out of range

OP:
----
b'\x01\x02\x03\x04\x05\x06\x07\x08\t\n'
3
9
7
6
```

bytearray Type:

--> It is a sequence data type, it able to represent sequence of numbers.

--> To represent numbers in bytearray type we have to use the following

two steps.
      1. Represent Numbers in List data type by using [] .
      2. Convert numbers from List type to bytearray type by using bytearray() fun.

EX:

```
list = [1,2,3,4,5]
print(list)
print(type(list))
ba = bytearray(list)
print(ba)
print(type(ba))
```

OP:

```
[1, 2, 3, 4, 5]
<class 'list'>
bytearray(b'\x01\x02\x03\x04\x05')
<class 'bytearray'>
```

--> bytearray type is able to allow the numbers from 0 to 255 only, if we provide numbers in out side range of 0 to 255 then PVM will provide any error like "ValueError: byte must be in range(0, 256)".

EX:

```
list = [253, 254, 255, 256]
ba = bytearray(list)
print(ba)
```

```
Status:  ValueError: byte must be in range(0, 256)
```

--> bytearray type objects are mutable objects, they are able to allow modifications on their content.

EX:

```
list = [1,2,3,4,5]
ba = bytearray(list)
for x in ba:
    print(x,end=" ")
print()
ba[2] = 100
for x in ba:
    print(x,end=" ")
```

OP:

```
1 2 3 4 5
1 2 100 4 5
```

--> In Python applications, we are able to read individual elements from bytearray on the basis of index values. If we provide any index value which is in out side rage of the bytearray index values then OVM will raise an error like "IndexError: bytearray index out of range"

In bytearray type, we are able to read elements in both forward direction and backward direction, to read elements in forward direction we have to use +ve index value and it must be started with 0 . To read elements in backward direction we have to use -ve index value and it must be started with -1.

```
EX:
---
list = [1,2,3,4,5,6,7]
ba = bytearray(list)
print(ba[3])
print(ba[-5])
print(ba[5])
print(ba[-4])
#print(ba[7]) --> IndexError: bytearray index out of range
print(ba[len(ba)-2])

OP:
---
4
3
6
4
6
```

In Python applications, to read elements from bytearray type we are able to use for loop and for-Each loop. To read elements in both forward direction and backward direction then we must use for loop only.If we use for-Each loop then we are able to read elements in forward direction only.

```
EX:
---
list = [1,2,3,4,5,6,7, 8, 9, 10]
ba = bytearray(list)
for x in range(0, len(ba)):
    print(ba[x],end=" ")
print()
for x in range(0, len(ba)):
    print(ba[-(x+1)],end=" ")

OP:
1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1

EX:
---
list = [1,2,3,4,5,6,7, 8, 9, 10]
ba = bytearray(list)
for x in ba:
    print(x,end=" ")

OP:
---
1 2 3 4 5 6 7 8 9 10
```

Q)What are the differences between bytes data type and bytearray data type?

Ans:
----
1. To rpresent elements in bytes data type we have to use a predefined function   like bytes().
   To represent elements in bytearray type we have to use a predefined function   like bytearray()

function   like bytearray().

2. bytes type objects are immutable objects.
   bytearray type objects are mutable objects.

Note: In Python, both bytes data type and bytearray data type are able to allow only numbers as elements and they are existed in fixed size in nature. If we are trying to access elements over its size then PVM will provide an error.

```
EX:
---
list = [1,2,3,4,5]
ba = bytearray(list)
for x in ba:
    print(x,end=" ")
#ba[5] = 6 --> IndexError: bytearray index out of range
#print(ba[5]) --> IndexError: bytearray index out of range
```

```
range type:
-----------
```

--> It is a sequence data type, it able to generate sequence of numbers.
--> To get numbers from range type we must use for loop.
--> To generate sequence of numbers from range type we have to use the following   syntaxes.
1. range(end)
--> It will generate sequence of numbers starts from 0 and upto the specified end-1 by using 1 step length.

```
EX:
---
r = range(10)
for x in r:
    print(x,end=" ")

OP:
0 1 2 3 4 5 6 7 8 9
```

2. range(Start, End)
--> It will generate sequence of numbers starts from the specified Start value and upto the Specified End-1 and by using 1 step length.

```
EX:
---
r = range(0,10)
for x in r:
    print(x,end=" ")
OP:
---
0 1 2 3 4 5 6 7 8 9

EX:
----
r = range(11,20)
for x in r:
    print(x,end=" ")
OP:
---
11 12 13 14 15 16 17 18 19
```

3. range(Start, End, Step)

```
--> It will generate sequence of numbers from the specified start value
and upto the specified end-1 value by using the specified step length.
EX:
----
r = range(0,20,2)
for x in r:
    print(x,end=" ")
Op:
0 2 4 6 8 10 12 14 16 18

List:
-----
--> It is a sequence data type, it able to represent sequence of
elements.
--> In List type, to represent elements we have to use [].
EX:
---
list = [1,2,3,4,5]
print(list)
print(type(list))
print(id(list))
OP:
[1, 2, 3, 4, 5]
<class 'list'>
3204551692872

--> List type is not in fixed size nature, it is existed with
dynamically growable nature, that is, it able to increase or decrease
its size depending on the no of elements which are existed in List.
EX:
---
list = [1,2,3,4]
print("List Elements :",list)
print("Length : ",len(list))
print()
list.append(5)
list.append(6)
print("List Elements :",list)
print("Length : ",len(list))
print()
list.append(7)
list.append(8)
print("List Elements :",list)
print("Length : ",len(list))
print()
list.remove(8)
list.remove(7)
print("List Elements :",list)
print("Length : ",len(list))
OP:
---
List Elements : [1, 2, 3, 4]
Length :  4

List Elements : [1, 2, 3, 4, 5, 6]
Length :  6

List Elements : [1, 2, 3, 4, 5, 6, 7, 8]
Length :  8
```

```
List Elements : [1, 2, 3, 4, 5, 6]
Length :  6
```

In Python List type is able to allow heterogeneous elements.
EX:

```
list = [1,2,3,4]
print(list)
list.append(22.22)
list.append(33.33)
print(list)
list.append(True)
list.append(False)
print(list)
list.append("abc")
list.append("xyz")
print(list)
```
OP:

```
[1, 2, 3, 4]
[1, 2, 3, 4, 22.22, 33.33]
[1, 2, 3, 4, 22.22, 33.33, True, False]
[1, 2, 3, 4, 22.22, 33.33, True, False, 'abc', 'xyz']
```

--> In List data type, we are able to read elements individually by using index values. In List type, it is possible to read elements in both forward direction and Backward direction, to read elements in forward direction we have to provide +ve index valuees and it must be started with 0 , to read elements in backward direction we have to provide -ve index values and it must be started with -1.
EX:

```
list = [1,2,3,4,5,6,7]
print(list)
print(list[2])
print(list[-4])
print(list[5])
print(list[-6])
print()
for x in range(0,len(list)):
    print(list[x], end=" ")
print()
for x in range(0,len(list)):
    print(list[-(x+1)], end=" ")
print()
for x in list:
    print(x, end=" ")
```
OP:

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

3
4
6
2

1 2 3 4 5 6 7
7 6 5 4 3 2 1
```

```
7 6 5 4 3 2 1
1 2 3 4 5 6 7
```

--> List is able to allow duplicate elements.
EX:

```
list = ["AAA", "BBB", "CCC", "DDD", "EEE"]
print(list)
list.append("BBB")
list.append("DDD")
print(list)
```
OP:

```
['AAA', 'BBB', 'CCC', 'DDD', 'EEE']
['AAA', 'BBB', 'CCC', 'DDD', 'EEE', 'BBB', 'DDD']
```

--> List is following insertion order, it is not following Sorting order.
EX:

```
list = ["AAA", "FFF", "BBB", "EEE", "CCC", "DDD"]
print(list)
```
OP:

```
['AAA', 'FFF', 'BBB', 'EEE', 'CCC', 'DDD']
```

--> List is able to allow None elements in any number.
EX:

```
list = ["AAA", "BBB", "CCC", "DDD", None, None, None]
print(list)
```

OP:

```
['AAA', 'BBB', 'CCC', 'DDD', None, None, None]
```

--> List objects are mutable objects, they are able to allow modifications on their content.
EX:

```
list = ["AAA", "BBB"]
print(list)
print(id(list))
list.append("CCC")
list.append("DDD")
print(list)
print(id(list))
list.append("EEE")
list.append("FFF")
print(list)
print(id(list))
```
OP:

```
['AAA', 'BBB']
2879331586632
['AAA', 'BBB', 'CCC', 'DDD']
2879331586632
['AAA', 'BBB', 'CCC', 'DDD', 'EEE', 'FFF']
2879331586632
```

Tuple Type:

--> It is a sequence type, it able to store sequence of elements.
--> It is index based, it able to allow to read elements on the basis of index values, It allows to read elements in both forward direction and backward direction, to read elements in forward direction we have to use +ve idex values and it must be started with 0, to read elements in Backward direction we have to provide -ve index values and it must be started with -1.
--> It allows duplicate elements.
--> It allows heterogeneous elements.
--> It is following insertion order.
--> It does not follow Sorting order.
--> Its objects are immutable objects.
--> It allows any no of None elements.

EX:

```
tuple = (1,2,3,4, 2, 5, None, "AAA", "BBB", None)
print(tuple)
print(type(tuple))
print(tuple[2])
print(tuple[-2])
#tuple[1] = "XXX"--> TypeError: 'tuple' object does not support item
assignment
```

OP:

```
(1, 2, 3, 4, 2, 5, None, 'AAA', 'BBB', None)
<class 'tuple'>
3
BBB
```

Q)What are the differences between List and Tuple?

Ans:
----

1. In List , all the elements are represented with [].
   In Tuple , all the elements are rpresented with ().

2. List is having append() and remove() functions to perform manipulations over the elements.
   Tuple is not having append() and remove() functions to perform manipulations over the elements.

3. List Objects are Mutable
   Tuple Objects are Immutable.

4. List is dynamically Gorwable in nature.
   Tuple is fixed size in nature.

set Type:

--> It is a sequence data type, it able to store sequence of elements.
--> To represent elements in set type we have to use {}.

EX:

```
set = {"AAA", "BBB", "CCC", "DDD", "EEE", "FFF"}
print(set)
print(type(set))
OP:
---
{'BBB', 'CCC', 'DDD', 'FFF', 'EEE', 'AAA'}
<class 'set'>

--> Set is not following both insertion order and Sorting order.
EX:
---
set = {"AAA", "BBB", "CCC", "DDD"}
print(set)
OP:
---
{'BBB', 'AAA', 'CCC', 'DDD'}

--> Set is not index based, if we are trying to read elements on the
basis of index     values then PVM will provide an error like
         "TypeError: 'set' object is not subscriptable"
EX:
---
set = {"AAA", "BBB", "CCC", "DDD"}
print(set)
#print(set[1]) --> Error
Status: TypeError: 'set' object is not subscriptable

--> To read elements from set we have to use for-Each loop, not normal
for loop.
EX:
---
set = {"AAA", "BBB", "CCC", "DDD"}
print(set)
for x in set:
    print(x)
OP:
---
{'AAA', 'CCC', 'BBB', 'DDD'}
AAA
CCC
BBB
DDD

--> Set is not allowing duplicate elements.
EX:
---
set = {"AAA", "BBB", "CCC", "DDD", "BBB", "CCC"}
print(set)

OP:
---
{'AAA', 'CCC', 'BBB', 'DDD'}

--> Set is able to allow heterogeneous elements.
EX:
---
set = {"AAA", "BBB", 10, 20, 22.22, 33.33, True, False}
print(set)
OP:
---
{False, 33.33, True, 'AAA', 10, 20, 22.22, 'BBB'}
```

EX:

```
set = {"AAA", "BBB"}
print(set)
set.add(10)
set.add(20)
print(set)
set.add(22.22)
set.add(33.33)
print(set)
set.add(True)
set.add(False)
print(set)
```

Op:

```
{'BBB', 'AAA'}
{'BBB', 10, 'AAA', 20}
{'BBB', 33.33, 10, 'AAA', 20, 22.22}
{False, 'BBB', 33.33, True, 10, 'AAA', 20, 22.22}
```

--> Set type objects are mutable objects, they are able to allow modifications on their content.
EX:

```
set = {10, 20}
print(set)
set.add(30)
set.add(40)
print(set)
set.add(50)
set.add(60)
print(set)
```

OP:

```
{10, 20}
{40, 10, 20, 30}
{40, 10, 50, 20, 60, 30}
```

--> Set is able to allow None element[Only one, not more than one].
EX:

```
set = {10, 20, None, None}
print(set)
```

OP:
```
{10, 20, None}
```

Q)What are the differences between List and Set?

Ans:
----
1. List is able to represent elements in the form of [].
   Set is able to represent all the elements in the form of {}.

2. List is index based.
   Set is not index based.

Set is not index based.

3. List is able to allow duplicate elements.
   Set is not allowing duplicate elements.

4. List is following insertion order.
   Set is not following insertion order.

5. List is able to allow None elements in any number.
   Set is able to allow only one None element.

6. List has append() function to add elements.
   Set has add() function to add elements.

7. List allows both for loop and for-Each to read elements.
   Set allows only for-Each loop to read elements.

8. Immutable form of List is Tuple.
   Immutable form of Set is FrozenSet.

FrozenSet Type

--> It is a Sequence data type, it able to represent sequence of elements.
--> To represent elements in frozenset type we have to use the following steps.
    1. Prepare Elements in Set type by using {}.
    2. Convert all the elements from Set type to frozenset type by using
       frozenset() predefined function.
--> It is not index based.
--> It is not following insertion order and sorting order.
--> It is not allowing duplicate elements.
--> It allows only one None element.
--> It allows heterogeneous elements.
--> It allows for-Each loop to read elements.
--> Its Objects are immutable objects.

EX:

```
set = {10, 20, 30, 40, 50, 20, 30, "AAA", "BBB", 22.22, 33.33, None, None}
fs = frozenset(set)
print(fs)
print(type(fs))
for x in fs:
    print(x,end=" ")
```

Q)What are the differences between Set and frozenset?

Ans:
----
1. To represent elements in Set we will use {}.
   To repersent elements in frozenset we must represent elelments in set by using {} then we must convert that elements into frozenset by using frozenset() function.

2. set objects are mutable objects.
   frozenset objects are immutable objects.

```
3. set is in dynamically growable in nature.
   frozenset is in fixed size in nature.

4. add() and remove() functions are existed in Set type.
   add() and remove() functions are not existed in frozenset type.

dict Type:
----------
--> It is a sequence data type, it able to store sequence of elements
in the form     of Key-Value pairs.
--> To represent elements in dict data type we will use the following
pattern.
    {Key1:Val1, Key2:Val2,....Key-n:Val-n}
EX:
---
stud_Dict = {111:'AAA', 222: 'BBB', 333:'CCC', 444:'DDD'}
print(stud_Dict)
print(type(stud_Dict))
print(id(stud_Dict))
OP:
---
{111: 'AAA', 222: 'BBB', 333: 'CCC', 444: 'DDD'}
<class 'dict'>
2171314711912

--> In dict type, duplicate elements are not possibl at keys side, but,
duplicate elements are possible at values side.
EX:
---
dict = {111:'AAA', 222: 'BBB', 333:'CCC', 444:'DDD', 222:'XXX',
555:'CCC'}
print(dict)
OP:
---
{111: 'AAA', 222: 'XXX', 333: 'CCC', 444: 'DDD', 555: 'CCC'}

--> It is not index based, but, we can get values on the basis of keys.
EX:
---
dict = {1:111, 2:222, 3:333, 4:444}
print(dict)
print(dict[1])
print(dict[2])
OP:
---
{1: 111, 2: 222, 3: 333, 4: 444}
111
222


--> It does not follow sorting order, but, it follows insertion order
w.r.t the keys.
EX:
---
dict = {'a':'AAA', 'f':'FFF', 'b': 'BBB', 'e':'EEE', 'c':'CCC',
'd':'DDD'}
print(dict)
OP:
---
```

```
{'a': 'AAA', 'f': 'FFF', 'b': 'BBB', 'e': 'EEE', 'c': 'CCC', 'd':
'DDD'}

--> Dict type is able to allow heterogeneous elements at both keys side
and values side.
EX:
----
dict = {1:111, 22.22:23456.3456, False:True, 'a':'AAA', 100:'XXX'}
print(dict)
OP:
----
{1: 111, 22.22: 23456.3456, False: True, 'a': 'AAA', 100: 'XXX'}

--> Dict data type is able to allow only one None element at keys side,
but, any no of None elements at values side.
EX:
----
dict = {None: 'AAA', None:'BBB', 1:None, 2:None}
print(dict)
OP:
---
{None: 'BBB', 1: None, 2: None}

3. Type Casting:
----------------
The process of converting data from one data type to another data type
is called as Type Casting.

To perform type casting Python has provided a set of predefined
functions.
1. int()
2. float()
3. complex()
4. bool()
5. str()

1. int():
----------
It can be used to convert the numbers from all the data types to int
type except complex type.
EX:
---
a = 22.22
str = '10'
bool = True
complex = 10+2j

float_To_Int = int(a)
print(float_To_Int)

str_To_Int = int(str)
print(str_To_Int)

bool_To_Int = int(bool)
print(bool_To_Int)

#complex_To_Int = int(complex) --> TypeError: can't convert complex to
int
#print(complex_To_Int)
```

```
#print(complex_To_Int)
```

OP:

```
22
10
1
```

2. float():

--> It can be used to convert data from all the data types to float data type except from Complex.

Note: if we are trying to convert complex number to float number by using float() function then PVM will generate an error like "TypeError: can't convert complex to float"

EX:

```
a = 10
b = True
str = '22.22'
c = 10+2j

int_To_Float = float(a)
print(int_To_Float)

bool_To_Float = float(b)
print(bool_To_Float)

str_To_Float = float(str)
print(str_To_Float)

#complex_To_Float = float(c) --> TypeError: can't convert complex to float
#print(complex_To_Float)
```

OP:
```
10.0
1.0
22.22
```

3. complex()

It can be used to convert data from all the data types to Complex data type.

Note: While converting data from int to complex, float to complex , bool to complex and str to complex then the provided values will be placed at real part in the generated complex number, in the generated complex number 0 will be provided as imaginary part.
EX:

```
a = 10
b = 22.22
c = True
str = '20'

int_To_Complex = complex(a)
```

```
print(int_To_Complex)

float_To_Complex = complex(b)
print(float_To_Complex)

bool_To_Complex = complex(c)
print(bool_To_Complex)

str_To_Complex = complex(str)
print(str_To_Complex)
```

OP:

```
(10+0j)
(22.22+0j)
(1+0j)
(20+0j)
```

If privide complex number in the form of String while converting data from String type to complex number then the provided complex number will be converted as it is from String type to complex type.
EX:

```
str1 = "10"
str1_To_Complex = complex(str1)
print(str1_To_Complex)# 10+0j

str2 = "20+5j"
str2_To_Complex = complex(str2)
print(str2_To_Complex)# 20+5j   or  (20+5j)+0j
```

OP:

```
(10+0j)
(20+5j)
```

4. bool():

It can be used to convert data from all the data types to bool data type.
EX:

```
a = 100
b = 22.22
c = 10 + 5j
d = "abc"

int_To_Bool = bool(a)
print(int_To_Bool)

float_To_Bool = bool(b)
print(float_To_Bool)

complex_To_Bool = bool(c)
print(complex_To_Bool)

str_To_Bool = bool(d)
print(str_To_Bool)
```

```
OP:
---
True
True
True
True
```

case-1: While converting data from int to bool , if the provided int value is non zero [+ve values and -ve values] then bool() function will return True value, if the provided int value is 0 then bool function will return 'False' value.

```
EX:
---
a = -100
b = 0
c = 100
print(bool(a))
print(bool(b))
print(bool(c))
OP:
---
True
False
True
```

Note: Same above situation is existed in the case of float to bool conversion.

```
EX:
---
a = -22.22
b = 0.0
c = 33.33
print(bool(a))
print(bool(b))
print(bool(c))
OP:
---
True
False
True
```

Case-2: In Python applications, it is possible to convert data from None type to bool type, always, it will provide 'False' as an output.

```
EX:
---
a = None
print(bool(a))
OP:
----
False
```

Case-3: While converting data from str type to bool type , if we provide any data as String then bool() function will return 'True' value, but, if we provide empty string[""] then bool() function will return 'False' value.

```
EX:
---
a = "0"
b = "100"
```

```
b = "100"
c = "abc"
d = " "#space
e = ""#No space
f = "False"
print(bool(a))
print(bool(b))
print(bool(c))
print(bool(d))
print(bool(e))
print(bool(f))
OP:
---
True
True
True
True
False
True
```

5. str()

The main intention of this function is to convert data from all the data types to str type.

```
EX:
---
a = 10
b = 22.22
c = 10+5j
d = None
e = True

int_To_Str = str(a)
print(int_To_Str)
float_To_Str = str(b)
print(float_To_Str)
complex_To_Str = str(c)
print(complex_To_Str)
none_To_Str = str(d)
print(none_To_Str)
bool_To_Str = str(e)
print(bool_To_Str)

OP:
---
10
22.22
(10+5j)
None
True
```

4. Python Statements

Statement is the collection of expressions.

To prepare Python applications, Python has provided the following type of statements.

1. General Purpose Statements

```
2. Input And Output Statements
3. Conditional Statements
4. Iterative Statements
5. Transfer Statements

1. General Purpose Statements:
-------------------------------
These Python statements are very much common and very much frequent in
Python applications.
EX:
Declaring variables
Declaring functions
Declaring Classes
Creating Objects for the classes
Accessing variables and functions
....
----

2. Input And Output Statements:
-------------------------------
Input:
------
If we use any statement to provide input data to the Python
applications then that statements are called as "Input Statements".

In Python applications, we are able to provide three types of input
data to the python programs.

1. Static Input
2. Dynamic Input
3. Command Line Input / Command Line Arguments

1. Static Input:
----------------
If we provide input data to the python applications at the time of
writing programs then that input data is called as Static input.
EX:
---
a = 10
b = 5
print("ADD :", (a+b))
print("SUB :", (a-b))
print("MUL :", (a*b))
OP:
---
15
5
50

2. Dynamic Input:
-----------------
If we provide input data to the python applications at Runtime then
that input data is called as Dynamic Input.

To take dynamic input in python applications, we have to use the
following predefined functions.

1. row_input()
2. input()
```

2. input()

Q)What is the differernce between row_input() and input() functions?

Ans:

In Python2.x version, row_input() can be used to read all the types of dynamic input and it will return the dynamic input in the form of String. In Python2.x version input() function can be used to read any type of dynamic input and it will return that dynamic input in the provided format like int, float, bool, ...

In python3.x version, row_input() function is not existed, only input() function is existed and it is same as row_input() function of Python2.x version, that is, input() function is able to read all the types of dynamic input but it able to return in the form of String type.
EX:

```
str = input("Enter Data : ")
print(str)
print(type(str))
```
OP:

```
Enter Data : Durga Software Solution
Durga Software Solution
<class 'str'>
```

In Python , input() function is able to read data in the form of String type even we enter the data of types int, float, bool,..... If we want to get dynamic input data in the form of the data types like int, float, bool,... then we have to use the following steps.

1. Read Dynamic input in the form of String by using input() function.
2. Convert data from String type to other data types like int, float, bool,.... by   using the predefined functions like int(), float(), bool(),...

EX:

```
a = input("First value  : ")
b = input("Second Value : ")
fval = int(a)
sval = int(b)
print("a   :",a)
print("b   :",b)
print("fval:", fval)
print("sval:", sval)
print("ADD :", (fval + sval))
print("SUB :", (fval - sval))
print("MUL :", (fval * sval))
```

OP:

```
First value  : 10
Second Value : 5
a   : 10
b   : 5
fval: 10
sval: 5
```

```
ADD : 15
SUB : 5
MUL : 50
```

EX:

```
eno = int(input("Employee Number  : "))
ename = input("Employee Name    : ")
esal = float(input("Employee Salary  : "))
eaddr = input("Employee Address : ")
empType = bool(input("Employee is Temp : "))
print()
print("Employee Details")
print("---------------------")
print("Employee Number   :", eno)
print("Employee Name     :", ename)
print("Employee Salary   :", esal)
print("Employee Address  :", eaddr)
print("Temporary Employee:", empType)
```
OP:

```
Employee Number  : 111
Employee Name    : AAA
Employee Salary  : 50000.0
Employee Address : Hyd
Employee is Temp :  [To give False value then dont provide any data]

Employee Details
---------------------
Employee Number   : 111
Employee Name     : AAA
Employee Salary   : 50000.0
Employee Address  : Hyd
Temporary Employee: False
```

Q)Is it possible to take more than one dynamic input by using single input() function?

Ans:

No, it is not possible to take more than one dynamic input from console by using single input() function. If we provide multiple values at a time on console then single input() function will read all the multiple values as single dynamic input of type str.

EX:

```
str = input("Enter Multiple Values : ")
print("str :",str)
print(type(str))
```
OP:

```
Enter Multiple Values : 10 20 30
str : 10 20 30
<class 'str'>
```

Note: If we want to get multiple values from our single dynamic input then we have to perform split operation over the dynamic input and get multiple values in the form of List and assign multiple values to

multiple variables.
EX:

```
str = input("Enter Multiple Values : ")
list = str.split()
a,b,c = list   # a,b,c = ['10','20','30']
print(int(a))
print(int(b))
print(int(c))
```
OP:

```
Enter Multiple Values : 10 20 30
10
20
30
```
EX:

```
a,b,c = [int(x) for x in input("Enter Multiple Values : ").split()]
print(a)
print(b)
print(c)
```

OP:

```
Enter Multiple Values : 10 20 30
10
20
30
```

EX:

```
a,b = [int(x) for x in input("Enter two Values : ").split()]
print("ADD :", (a+b))
print("SUB :", (a-b))
print("MUL :", (a*b))
```
OP:

```
Enter two Values : 10 5
ADD : 15
SUB : 5
MUL : 50
```

If we want to evaluate the expressions which we provided as dynamic input then we have to use eval() function.
Note: input() function will read the provided expression as single dynamic input of type Str and it is not performing any evaluations over the provided expression.
EX:

```
expr = eval(input("Enter Expression : "))
print(expr)
```
OP:

```
Enter Expression : 10+20*3
70
```

Q)Is it possible to read list / tuple / set of values as dynamic input directly?

Ans:

No, input() is able to read all the types of data as String. If we want to get List, Tuple, Set, dict of elements as dynamic input then we have to use the following steps.
1. Read list / tuple/ set/ dict type elements as dynamic input in the form of   String by using input() function.
2. Convert String dynamic input into List, tuple, set and dict by using eval()   function.
EX:

```
list = eval(input("Enter List Elements with [] : "))
tuple = eval(input("Enter Tuple Elements with or with out () : "))
set = eval(input("Enter Set Elements with {} : "))
dict = eval(input("Enter dict Elements with {k:v} : "))

print(list, "----->", type(list))
print(tuple, "----->", type(tuple))
print(set, "----->", type(set))
print(dict, "----->", type(dict))
```

OP:
```
Enter List Elements with [] : [10,20,30,40,50]
Enter Tuple Elements with or with out () : (10,20,30,40,50)
Enter Set Elements with {} : {10,20,30,40,50}
Enter dict Elements with {k:v} : {10:100,20:200,30:300,40:400,50:500}
[10, 20, 30, 40, 50] -----> <class 'list'>
(10, 20, 30, 40, 50) -----> <class 'tuple'>
{40, 10, 50, 20, 30} -----> <class 'set'>
{10: 100, 20: 200, 30: 300, 40: 400, 50: 500} -----> <class 'dict'>
```

3. Command Line Input / Command Line Arguments:

If we provide input data to the python program by specifying values along with "python" or "py" command on console or command prompt then that input data is called as Command Line input or Command Line Arguments.

EX: `D:\python10>python Add.py 10 20`

If we use the above command on command prompt then PVM will perform the following actions.
1. PVM will read all the command line input including python file name from console
2. PVM will store all the command line input in the form of String in a list   refered by argv in sys module.
Note: To get elements from 'argv' list type , we have to import argv in the present python file.

```
     from sys import argv
```

EX:

Test.py

```
from sys import argv
print(argv)
```

```
for x in argv:
        print(x)

OP:
D:\python10>python Test.py 10 20 30
['Test.py', '10', '20', '30']
Test.py
10
20
30

EX:Cal.py
---------
from sys import argv
a = int(argv[1])
b = int(argv[2])
print("ADD :",(a+b))
print("SUB :",(a-b))
print("MUL :",(a*b))

D:\python10>py Cal.py 10 5
ADD : 15
SUB : 5
MUL : 50

EX: Test.py
------------
from sys import argv
print("No Of Command Line Arguments :",len(argv)-1)
sum = 0
print("Command line Arguments List  :",end=" ")
for x in range(1,len(argv)):
        print(argv[x],end=" ")
        sum = sum + int(argv[x])
print()
print("SUM of Command Line Arguments:",sum)

OP:
D:\python10>py Test.py 10 20 30 40
No of Command Line Arguments: 4
Command Line Input List : 10 20 30 40
SUM of Command Line Arg : 100
```

In Command line Inputs case, PVM will read all the command line inputs from console on the basis of space seperator. If we want to provide a String data as comand line input including multiple multiple words then we have to use " "[Double quotations], it is not possible to use Single quotations and triple quotations.

```
EX:Test.py
----------
from sys import argv
print(argv)
print(len(argv)-1)

OP:
---
D:\python10>py Test.py Durga Software Solutions
['Test.py', 'Durga', 'Software', 'Solutions']
3
```

```
3

D:\python10>py Test.py 'Durga Software Solutions'
['Test.py', "'Durga", 'Software', "Solutions'"]
3

D:\python10>py Test.py "Durga Software Solutions"
['Test.py', 'Durga Software Solutions']
1

D:\python10>py Test.py '''Durga Software Solutions'''
['Test.py', "'''Durga", 'Software', "Solutions'''"]
3
```

Output Statements:

If we send data from Python applications to output devices like Console, Printer, Network,..... then that Data is called as Output Data and this operation is called as Output Operation.

To perform Output operation if we use any statement then that statement is called as Output Statement.

To perform out put operation in python applications we will use a predefined function in the form of print(--), Where print() function was defined in such a way to display the provided data or the provided variables values and to keep cursor in the next line.

EX:

```
std_Roll_No = int(input("Student Roll Number   : "))
std_Name = input("Student Name           : ")
std_Aggr = float(input("Student Aggregate     : "))
std_Qual = eval(input("Student Qualifications : "))
std_Marks = eval(input("Student Marks         : "))
std_Courses = eval(input("Student Courses  : "))
std_Year_Of_Passes = eval(input("Student Year Of Passes : "))
print()
print("Student Details")
print("---------------------")
print("Student Roll Number   :", std_Roll_No)
print("Student      Name     :", std_Name)
print("Student Aggregate %   :", std_Aggr)
print("Student Qualifications:", std_Qual)
print("Student Marks         :", std_Marks)
print("Student Courses       :", std_Courses)
print("Student Year of Passes:", std_Year_Of_Passes)
```

OP:

```
Student Roll Number   : 111
Student Name          : Durga
Student Aggregate     : 85.5
Student Qualifications : ["BTech","MTech","PHD"]
Student Marks        : (66,67,89,87,69,77)
Student Courses  : {"JAVA", "Python", "Oracle"}
Student Year Of Passes : {"BTECH":2000,"MTECH":2002,"PHD":2007}

Student Details
```

```
---------------------
Student Roll Number   : 111
Student  Name         : Durga
Student Aggregate %   : 85.5
Student Qualifications: ['BTech', 'MTech', 'PHD']
Student Marks         : (66, 67, 89, 87, 69, 77)
Student Courses       : {'JAVA', 'Oracle', 'Python'}
Student Year of Passes: {'BTECH': 2000, 'MTECH': 2002, 'PHD': 2007}
```

In print() function, we are able to perform concatination operation by using , and + .

To perform concatination in print() function if we use + operator then we must provide both the data of String type, it is not possible to perform concatination between String and other data types.

```
EX:
age = 25
print("Age :"+age)
Status: TypeError: can only concatenate str (not "int") to str

EX:
name = "Durga"
print("Name :"+name)
OP:
Name :Durga

EX:
print("Durga"+"Software"+"Solutions")
OP:DurgaSoftwareSolutions
```

To perform concatination operation in print() function if we use , then we are able to get the concatination result with space seperator and it is possible to perform concatination between String type and any other type including string, int, float, bool,....

```
EX:
---
name = "Durga"
age = 25
salary = 25000.0
print("Name   :",name)
print("Age    :",age)
print("Salary :",salary)
OP:
---
Name   : Durga
Age    : 25
Salary : 25000.0

EX:
---
print("Durga","Software","Solutions")

OP:
---
Durga Software Solutions
```

In Print() function, if we want to concatinate a particular value to the string at the end and to keep cursor at the same line we have to use "end" keyword.

-
EX:

```
seconds = 45
minuts = 50
hours = 10
day = "Mon Day"
date = 15
month = 8
year = 2019
print("Today Date And Time ",end=":")
print(day,end="  ")
print(date,end=":")
print(month,end=":")
print(year,end="  ")
print(hours,end=":")
print(minuts,end=":")
print(seconds)
```

OP:

```
Today Date And Time :Mon Day  15:8:2019  10:50:45
```

In print() function, if we want to display multiple values with a particular seperator then we have to use "sep" keyword.
EX:

```
a = 10
b = 20
c = 30
d = 40
print("a,b,c,d values :",a,b,c,d,sep=" ")
```

OP:

```
a,b,c,d values : 10 20 30 40
```

EX:

```
seconds, minuts, hours, day, date, month, year = 45, 50, 10, "Monday",
15, 8, 2019
print("Today Date And Time ",end=":")
print(day,end="  ")
print(date,month,year,sep="/",end="  ")
print(hours,minuts,seconds,sep=":")
```
OP:

```
Today Date And Time :Monday  15/8/2019  10:50:45
```

In print() function , if we want to perform concatination over a particular string upto the specified number of times then we have to use '*' notation.
Syntax:

```
1. print("str"*n) --> It will perform concatination upto n times
2. print(n*"str") --> It will perform concatination upto n times
3. print(n1*"str"*n2)--> It will perform concatination upto n1*n2 times
```

EX:

```
---
print("Durgasoft "*4)
print(4*"Durgasoft\t")
print(4*"Durgasoft\n"*4)

OP:
---
Durgasoft Durgasoft Durgasoft Durgasoft
Durgasoft       Durgasoft       Durgasoft       Durgasoft

Durgasoft
-----
--16 times-
```

In python applications, we are able to format the outputs by using '%' operator through print() function.

```
%i ---> int
%d ---> int
%s ---> str, list, tuple,....
%f ---> float
```

EX:

```
---
a = 10
b = "abc"
c = 22.22
print("a , b and c values are %d %s %f"%(a,b,c))

OP:
---
a , b and c values are 10 abc 22.220000
```

In general, we will use formatted output while preparing template messages as outputs.
Note: Template messages is a message with variables, where variables will have values at runtime.
EX:

```
----
name = input("Name       : ")
age = int(input("Age        : "))
company = input("Company    : ")
salary = float(input("Salary     : "))
skills =  eval(input("Skill Set :"))
print("I am %s, my age is %d years and i am working with %s  \n and My
Salary is %f and My Skill set is %s"%(name,age,company,salary,skills))

OP:
---
Name       : Anil
Age        : 32
Company    : TCS
Salary     : 55000.0
Skill Set :["Java", "Python", "Oracle", "UI Techs"]
I am Anil, my age is 32 years and i am working with TCS
and My Salary is 55000.000000 and My Skill set is ['Java', 'Python',
'Oracle', 'UI Techs']
```

In print() function we are able to format the data by using place holders also.
In python applications, w are able to provide place holders in print() function in the following two ways.
1. Index Based Place holders.
2. Name Based Place holders.

1. Index Based Place holders:

Syntax: {index}
To provide value to the place holder we have to use format() function with value.
EX:

```
a = 10
b = 20
c = 30
print("a value is {0} \nb value is {1} \nc value is {2}".format(a,b,c))
```

OP:

```
a value is 10
b value is 20
c value is 30
```

2. Name Based Place Holders.

Syntax: {name}
To provide value to the Name based place holder we have to use format() function with value assignment to the respective place holder name.
EX:

```
a = 10
b = 20
c = 30
print("a value is {x} \nb value is {y} \nc value is
{z}".format(x=a,y=b,z=c))
```
OP:

```
a value is 10
b value is 20
c value is 30
```

In Python applications, in print() function, we are able to provide both index based place holders and name based place holders , but, first we have to provide index based place holders next we have to provide name based place holder. If we provide name based place holders first and index based place holders next then PVM will raise an error like "Syntax Error: positional argument follows keyword argument".

EX:

```
a = 10
b = 20
c = 30
d = 40
print("a value is {0} \nb value is {1} \nc value is {x}\nd value is
{y}".format(a,b,x=c,y=d))
```
OP:

```
---
a value is 10
b value is 20
c value is 30
d value is 40

EX:
---
a = 10
b = 20
c = 30
d = 40
print("a value is {x} \nb value is {y} \nc value is {2}\nd value is
{3}".format(x=a,y=b,c,d))
Status: SyntaxError: positional argument follows keyword argument

3. Conditional Statements:
----------------------------
These statements are able to allow to execute a block of instructions
on the basis of a particular condition.
EX: if
Syntax-1:
---------
if condition:
  --instructions--

Syntax-2:
---------
if condition:
  ---instructions---
else:
  ---insrtuctions----

Syntax-3:
---------
if condition:
  ---instructions----
elif condition:
  ---instructions----
elif condition:
  ---instructions----
---
---
else:
  ---instructions----

EX:
---
name = input("Enter Name : ")
if name == "Durga":
    print("Your Name is %s"%name)

OP:
----
Enter Name : Durga
Your Name is Durga

EX:
```

```
---
name = input("Enter Name : ")
if name == "Durga":
    print("Your Name is Durga")
else:
    print("Your Name is not Durga, Your Name is %s"%name)

OP:
---
Enter Name : Anil
Your Name is not Durga, Your Name is Anil

EX:
---
no = int(input("Enter a number from 1 to 7 : "))
if no == 1:
    print("Monday")
elif no == 2:
    print("Tuesday")
elif no == 3:
    print("Wensday")
elif no == 4:
    print("Thursday")
elif no == 5:
    print("Friday")
elif no == 6:
    print("Saturday")
elif no == 7:
    print("Sunday")
else:
    print("Sorry, No week day for your number, please enter number from
1 to 7")

OP:
---
Enter a number from 1 to 7 : 5
Friday

OP:
---
Enter a number from 1 to 7 : 10
Sorry, No week day for your number, please enter number from 1 to 7

EX:
---
no = int(input("Enter a Number : "))
if no % 2 == 0 :
    print("%d is Even Number"%no)
else:
    print("%d is Odd NUmber"%no)

OP:
---
Enter a Number : 10
10 is Even Number

OP:
----
Enter a Number 5
```

```
5 is Odd Number

EX:
---
no1 = int(input("Enter First Number  : "))
no2 = int(input("Enter Second Number : "))
if no1 < no2:
    print("%d is Biggest Number"%no2)
elif no1 > no2:
    print("%d is Biggest Number"%no1)
else:
    print("Both are Equal")

OP:
---
Enter First Number  : 10
Enter Second Number : 20
20 is Biggest Number

OP:
---
Enter First Number  : 10
Enter Second Number : 10
Both are Equal

EX:
---
a = int(input("Enter First Number  : "))
b = int(input("Enter Second Number : "))
c = int(input("Enter Third Number  : "))

if a > b:
    if a > c:
        print("%d is Biggest Number"%a)
    else:
        print("%d is Biggest Number"%c)
elif b > c:
    if b > a:
        print("%d is Biggest Number"%b)
    else:
        print("%d is Biggest Number"%a)
elif c > a:
    if c > b:
        print("%d is Biggest Number" %c)
    else:
        print("%d is Biggest Number" %b)
else:
    print("All are Equal")

OP:
---
Enter First Number  : 10
Enter Second Number : 20
Enter Third Number  : 30
30 is Biggest Number

OP:
---
Enter First Number  : 10
Enter First Number  : 10
Enter Second Number : 10
Enter Third Number  : 20
20 is Biggest Number
```

OP:

```
Enter First Number  : 10
Enter Second Number : 10
Enter Third Number  : 10
All are Equal
```

Note: In C, C++ and Java switch programming consatruct is existed , but, in Python switch programming construct is not existed, we are able to manage switch programming construct by using if-elif-else syntax.

4. Iterative Statements:

These statements are able to allow to execute a set of instructions repeatedly on the basis of a particular conditional expression.
EX: for, while
Note: In Python do-while iterative statement is not existed.

for :

```
for var in range(start, end, step)/sequence_Data_Type:
   ----instructions------
```
EX:

```
for x in range(10):
    print(x)
```

EX:

```
for x in range(0,10):
    print(x)
```
EX:

```
for x in range(0,10,1):
    print(x)
```

OP:

```
0
1
2
3
4
5
6
7
8
9
```

EX:

```
for x in range(0,10,1):
    print(9-x)
```

```
OP:
---
9
8
7
6
5
4
3
2
1
0

EX:
---
no = int(input("Enter Number  : "))
for x in range(0,no,2):
    print(x)
OP:
---
0
2
4
6
8
10
12
14
16
18

EX:
---
no = int(input("Enter Number  : "))
for x in range(1,no,2):
    print(x)
OP:
---
1
3
5
7
9
11
13
15
17
19

EX:[Prime Number]
-----------------
no = int(input("Enter Number  : "))

for x in range(2,no+1):
    b = True
    for y in range(1,x):
        if not((y==1) or (y==x)) :
            if x%y == 0:
                b = False
    if b == True:
        print("%d is Prime Number"%x)
```

OP:

```
Enter Number  : 20
2 is Prime Number
3 is Prime Number
5 is Prime Number
7 is Prime Number
11 is Prime Number
13 is Prime Number
17 is Prime Number
19 is Prime Number
```

Note: Providing a loop inside a loop is called as nested loop.
EX:

```
for x in range(0, 10, 1):
    for y in range(0, 10, 1):
        print(x,"  ", y )
```
OP:

```
0  0
0  1
----
----
0  9
----
----
----
9  0
----
9  9
```

If we want to read elements from the data types like str, list, tuple, set, ... then we have to use for-Each loop.
EX:

```
str = "Durga Software Solutions"
for x in str:
    print(x,end="  ")
```

OP:

```
D  u  r  g  a     S  o  f  t  w  a  r  e     S  o  l  u  t  i  o  n  s
```

EX:

```
list = [10,20,30,40,50,60,70,80,90, 100]
for x in list:
    print(x,end="  ")
```

OP:

```
10  20  30  40  50  60  70  80  90  100
```

```
EX:
---
dict = {"A":"AAA", "B":"BBB", "C":"CCC", "D":"DDD", "E":"EEE",
"F":"FFF"}
for x in dict:
    print(x,"--->",dict[x])
OP:
---
A ---> AAA
B ---> BBB
C ---> CCC
D ---> DDD
E ---> EEE
F ---> FFF

while:
-------
Syntax:
-------
while condition:
  ----instructions-----

EX:
---
a = 0;
while a < 10:
    print(a)
    a = a + 1

OP:
---
0
1
2
3
4
5
6
7
8
9

EX:
---
str = "Durga Software Solutions"

a = 0;
while a < len(str):
    print(str[a])
    a = a+ 1
OP:
---
D
u
r
g
a

S
```

o
f
t
w
a
r
e

S
o
l
u
t
i
o
n
s

EX:

```
list = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
result = 0
a = 0
print("List of Elements :", end = " ")
while a < len(list):
    print(list[a],end="  ")
    result = result + list[a]
    a = a+ 1
print()
print("SUM of List Elements :", result)
print("AVG of all List Elements :",(result/len(list)))
```

OP:

```
List of Elements : 10  20  30  40  50  60  70  80  90  100
SUM of List Elements : 550
AVG of all List Elements : 55.0
```

EX:

```
dict = {"A":"AAA", "B":"BBB", "C":"CCC", "D":"DDD", "E":"EEE",
"F":"FFF"}
list = list(dict.keys())

a = 0;
while a < len(list):
    print(list[a],"---->",dict[list[a]])
    a = a + 1
```
OP:

```
A ----> AAA
B ----> BBB
C ----> CCC
D ----> DDD
E ----> EEE
F ----> FFF
```

5. Transfer Statements:

These statements are able to bypass flow of execution from one instruction to another instruction.

EX: break, continue

break:

In general, break statement will be used in loops, it will bypass flow of execution to out side of the current loop.

```
EX:
---
print("Before Loop")
for x in range(0,10):
    if x == 5:
        break
    print(x)
print("After Loop")
OP:
---
Before Loop
0
1
2
3
4
After Loop
```

In python applications, if we provide 'break' statement inside the nested loop then break statement will give effect to the respective nested loop only, not to outer loop.

```
EX:
---
for x in range(0,10):
    for y in range(0,10):
        if y == 5:
            break;
        print(x,"---->",y)
OP:
---
0 0
0 1
---
0 4
---
---
9 0
9 1
---
9 4
```

continue:

It will skip the remaining instructions execution in current iteration and it will continue with next iteration.

```
EX:
---
for x in range(0,10):
    if x == 5:
        continue
    print(x)
OP:
---
0
1
2
3
4
6
7
8
9
```

In Python applications, if we provide 'continue' statement inside nested loop then 'continue' statement will give effect to nested loop only, it will not give effect to outer loop.

```
EX:
----
for x in range(0,10):
    for y in range(0, 10):
        if y == 5:
            continue
        print(x,"---->",y)
OP:
---
0 ----> 0
0 ----> 1
0 ----> 2
0 ----> 3
0 ----> 4
0 ----> 6
0 ----> 7
0 ----> 8
0 ----> 9
-----
-----
9 ----> 0
9 ----> 1
9 ----> 2
9 ----> 3
9 ----> 4
9 ----> 6
9 ----> 7
9 ----> 8
9 ----> 9
```

pass:

It can be used to pass flow of execution to out side of the current block.

```
EX:
---
a = 20
if a == 10:
    print("a value is %d"%a)
else:
   pass
```

```
print("After if-else")

OP:
---
After if-else
```

del:

It can be used to delete a particular variable from Python program scope. If we delete any variables by using 'del' keyword then we are unable to use that variable, in the case of we use that variable then PVM will raise an error like "NameError: name 'a' is not defined".
EX:

```
a = 10
print("a value is %d"%a)
del a
print("a value is %d"%a)

OP
a value is 10
"NameError: name 'a' is not defined".
```