OPERATORS :

→ operators are Special charactors (or) Symbols each and every symbol perform Specific task.

→ It will perform in between two operants.

→ This operators returns boolean values (or) Actual values.

OPERANDS :

→ operands is nothing but a values (or) literals (or) Expression

→ Expression : It is tombination of values and operators.

→ It is a unprocessed statement.

SYNTAX

(1) operand 1 operator operand 2

(2) operand operator.

Classification of operators :

(1) Arthematical operators

(2) logical operators

(3) Relational operator (comparision and relation)

(4) Membership operator.

(5) Bit wise operator.

(6) Assignment operator

(7) Identity operator

(1) Arthematical operator:

→ TO perform the arthematical operation by the help of arthematical operators.

→ It returns actual values.

→ These arthematical operators are

(a) addition
(b) Subtraction
(c) Multiplication
(d) True division
(e) floar division.
(f) Modulous.
(g) Exponential (power operator)

(a) ADDITION:

→ TO find out the addition operator in between two (or) more values.

→ we represent addition operator as '+' symbol

→ Return actual value

Divided into two types (i) addition
(ii) con catation.
Catagation

→ If we perform addition operator in between two single value data type, we called as addition.

→ If we perform addition operator in b/w two Multi value data type, we called as con catagation.

Note: When we perform con catination both the operand value data types should be same.

Examples

→ 10 + 20 ⇒ 30

→ 10.05 + 50.06 ⇒ 60.11

→ (10+5j) + (10+5j) ⇒ (20+10j)

→ b'1010' + b'101' ⇒ b'10101101'

→ True + True ⇒ 2

→ 10 + 10.5 ⇒ 20.5

→ 10 + (10+5j) ⇒ 20+5j

→ 10 + True ⇒ 11

→ 10.06 + (10+5j) ⇒ (20.060000000000002+5j)

→ 10.06 + b'101' ⇒ Error unsupported operand

→ 10.06 + True ⇒ 11.06

→ (10+5j) + b'1010' ⇒ Error (unsupported operand)

→ (10+5j) + False ⇒ (10+5j).

Note: concatenation supports only for string, list and tuple

ex: "str" + "ing" ⇒ 'string'

"str" + 2 ⇒ Error (Not possible for str + int)

"str" + [10] ⇒ Error(" " " " + list)

"str" + (10,) ⇒ error(" " " " + tuple)

[10,20,30] + [40,50] ⇒ [10,20,30,40,50]

[10,20,30] + (10,) ⇒ Error(not possible for list + tuple)

(10,40,50) + (60,70) ⇒ (10,40,50,60,70)

{10,20} + {40,50} ⇒ Error (not support for set+set

{10:20} + {40:50} ⇒ error (" " " dict+dict

(b) Subtraction.

→ To perform the subtraction operator we perform/ find out the subtraction between 2 operands

→ we represent operator called '-' minus

→ It will support integer, float, complex and boolean, set

Examples

→ 50-10 ⇒ 10

→ 51.01 - 1.01 ⇒ 50

→ (10+5j) - (5+5j) ⇒ (5+0j)

→ b'101' - b'10' ⇒ Error (unsupport for byte - byte).

→ True - False ⇒ 1

→ 50 - 16.05 ⇒ 33.5

→ 50 - (60-5j) ⇒ (-10+5j)

→ 50 - True ⇒ 49

→ 50.06 - (5+5j) ⇒ 45.06 - 5j

→ 50.03 - True ⇒ 49.03

→ 10+5j - True ⇒ (9+5j)

→ "str" - "hai" ⇒ Error

→ [10,20] - [50,60] ⇒ Error

→ (10,20) - (50,60) ⇒ Error

→ {10,20} - {50,60} ⇒ {10,20}

→ {10,20} - {50,60,20} ⇒ {10}

⇒ {10:20} - {50:60} ⇒ Error.

MULTIPLICATION:

→ To perform the product of two operands.
→ Represent with '*'
→ It will support integer, float, compler, boolean.
→ classified into two types (i) Multiplication
(ii) Replication.

→ If we perform multiplication in between 2 single value types we called as Multiplication
→ If you perform multiply in between collection and integer we called as Replication.

Examples

10*20 ⇒ 200
10.05 *50.06 ⇒ 503.103000000000 07
(10+5j) * (5+6j) ⇒ (20+85j)
True * True ⇒ 1
10 * 10.05 ⇒ 100.5
10 * (15+5j) ⇒ (150+50j)
10 * True ⇒ 10
10.06 * (5+5j) ⇒ (50.300000000000 4 +50.30000004j)
10.06 * True ⇒ 10.06
(10+5j) * False ⇒ 0j
"str" * "str" ⇒ Error
[10,20] * [10,20] ⇒ Error
(10,20) * (10,20) ⇒ Error
{10,20} * {10,20} ⇒ Error
{10:20} * {10:20} ⇒ Error
"str" * 3 = 'strstrstr'
[10,20] * 3 = [10,20,10,20, 10,20]

(10,20) * 3 ⇒ (10,20,10,20,10,20)
{10,20} * 3 ⇒ Error
{10:20} * 3 ⇒ Error.

True DIVISION:

→ To find out the Quotent of inbetween 2 number.
→ we represent as forward slash '*' '/'
→ Support only single value data types except bytes
→ Returns only float value.

Examples

10/20 ⇒ 0.5

10.06/20.06 ⇒ 0.5014955134596212

(10+5j)/(2+3j) ⇒ (2.692

True / False ⇒ Error

False/True ⇒ 0.0

10/5.0 ⇒ 2.0

10/(10+5j) ⇒ (0.8−0.4j)

10/True ⇒ 10.0

10.05/(16+5j) ⇒ (0.57224199288−0.17882562277j)

12.06/True ⇒ 12.06

10+5j/6 ⇒ (10+0.8333333333333334j)

"Str"/j ⇒ Error

[10,20]/[10] ⇒ Error

(10,20)/(10,) ⇒ Error

{10,20}/{10} ⇒ Error

{10:20}/{30:40} ⇒ Error

Floor DIVISION

→ we will find out quotient of rounded value in between two values

→ Represent as '//'

→ when we pass the two integer values then it return rounded value (integer).

→ When we pass any other type of values then it return float values

→ supports only integer, float and boolean.

Examples

10 / 5 ⇒ 2.0

10 // 5 ⇒ 2

15.0 // 6 ⇒ 2.0

15.0 / 6 ⇒ 2.5

10+6j // 10+6j ⇒ Error

True // True ⇒ 1

"Str" // "Str" ⇒ Error

b'1010' // b'101' ⇒ Error

[10,20] // [10] ⇒ Error

(10,20) // (10) ⇒ Error

{10,20} // {10} ⇒ Error

{10:20} // {40:50} ⇒ Error.

8/01/23
Saturday

MODULOUS:

→ To find the remainder in between two operands
→ we represent as percentage '%' symbol
→ Support only single value data type except bytes and complex.

Examples

30 % 10 ⇒ 0
33 % 11 ⇒ 10
10.05 % 2.6 ⇒ 2.2500000000004
(10+5j) % (5+6j) ⇒ Error
b'1010' % b'10' ⇒ Error
False % True ⇒ 0
10 % 2.5 ⇒ 0.0
10.05 % True ⇒ 0.050000000000071
10 % True ⇒ Error.

POWER (OR) EXPONENTIAL:

→ To findout the power of given value. we have to use one base value and power operand.
→ we represent as '**'
→ It returns the boolean values.

Example

10**2 ⇒ 100
10.05**2 ⇒ 101.0025000000001
10+5j**2 ⇒ (-15+0j)

True ** 2 ⇒ 1

10 ** 10.06 ⇒ 1148153621 4.96

10.05 ** 1.6 ⇒ 16.080000000000002

10+5j ** 1.5 ⇒ (2.094305849 57.+7.905694150409j)

LOGICAL OPERATORS

→ we will perform the logical operation inbetween two operands.

→ To find out the logical operation in between two expressions.

→ In logical operator it return as ~~f~~actual value (or) boolean value.

→ It is used for findout the condition is satisfied (or) not in between two expressions

Classified into three types

(i) logical and operator

(ii) logical or operator

(iii) logical not operator.

(i) Logical and operator:

→ If the operand one condition (or) expression is satisified. then it return operand two value.

→ If the operand one condition (or) expression is not satisfied (False) then it retun operand one value

→ We represent as "and"

→ and is a key word as well as operator

→ It returns actual values

operand 1 and operand 2	result
True (NDF)	operand 2 value.
False (DF)	operand 1 value.

Examples

Expression → Actual value, bool value

Actual value
- Default → False
- Non default → True

0 and 1 ⇒ 0

1 and 0 ⇒ 0

1 and 1 ⇒ 1

0 and 0 ⇒ 0

0 and 0.0 ⇒ 0

0.0 and 0 ⇒ 0.0

20-20 and 30-15 ⇒ 0

30-15 and 20-20 ⇒ 0

30-15 and 20-10 ⇒ 10

20-10 and 30-15 ⇒ 15

[10,20] and "" ⇒ ""

[10,20] and {'a':10} ⇒ {'a':10}

10 and 60 ⇒ 60

"" and (10,50) ⇒ ""

(10,20) and {10,20} ⇒ {10,20}

{10,20} and {} ⇒ {}

10 and 20 and 30 ⇒ 30

(10 and 20) and 30 ⇒ 30

(10 or 20) and 30 ⇒ 30

(ii) LOGICAL OR OPERATOR:

→ If the operand one condition (or) Expression is satisfied then it returns operand one value

→ If the operand one condition (or) Expression is not satisfied then it return operand two value

→ we represent as 'or'

→ or is a key word as well as operator.

→ It returns the actual value.

operand 1 or operand 2	result
True (OPV)	Operand 1 value/result
False (OF)	operand 2 value/result

Examples

Default	Non Default
False	True

0 or 0 ⇒ 0

0 or 1 ⇒ 1

1 or 0 ⇒ 1

1 or 1 ⇒ 1

10 or 20 ⇒ 10

20 or 10 ⇒ 20

10 or " " ⇒ 10

" " or 10 ⇒ 10

( ) or [ ] ⇒ [ ]

[ ] or " " ⇒ ( )

{10:20} or "str" ⇒ {10:20}

"str" or [10, 50] ⇒ 'str'

50-90 or 50-30 ⇒ -40

10 or 20 and 30 ⇒ 10

LOGICAL NOT OPERATOR

→ If operand one is satisfied then it return False

→ If operand one is not satisfied then it return True

→ we represent as 'not'

→ not is a key word as well as operator.

→ It return only boolean values.

SYNTAX

→ not (operand)

(or)

→ not operand.

operand	Result
True	False
False	True

Examples

not True ⇒ False

not False ⇒ True

not 0 ⇒ True

not 10 ⇒ False

not [10,20] ⇒ False

not [ ] ⇒ True

not {10,20} ⇒ False

not { } ⇒ True

10 and 20 and 30 ⇒ 30

(10 and 20) and 30 ⇒ 30

10 or 20 and 30 ⇒ 10

(10 or 20) and 30 ⇒ 30

not ([10,20] and {'a':10}) or {10,20} ⇒ {10,20}

not ([10,20] and { }) or {10,20} ⇒ True.

[10,20] and { } or {10,20} ⇒ {10,20}.

→ (10+20*60 and 10**2) or ({10,20} or not [15-16])

(1210 and 100) or ({10,20} or False)

(100) or ({10,20})

⇒ 100

COMPARICION OPERATOR.

→ TO perform comparicion in between two or more operands

→ Internally it check the waitage of the values for single value data type.

→ It will checks the type of the (collection value) and waitage of the values. for multi value data type

→ It return boolean values

Classified into two types

(i) Equalto Equalto (==)

(ii) not Equal (!=)

(i) Equal to Equal to

→ Represent as "= ="

→ In single value data type it will check the waitage of the between two values.

→ If values are same / waitage is same it return True

→ If values / waitage not same then it return False

→ For multivalue it check the type of the collection and waitage of the values.

Syntax

operand 1 = = operand 2

→ 10 = = 20 ⇒ False

10 = = 10.0 ⇒ True

10+5j = = 10+6j ⇒ False

10+5j = = 10+5.0j ⇒ True

True = = True ⇒ True

"Str" = = "Str" ⇒ True

"AB" = = "AC" ⇒ False

'ab' = = 'ba' ⇒ False

[10,20] = = [10,20] ⇒ True

[10,20] = = [20,10] ⇒ False

[10,20] = = (10,20) ⇒ False

[10,20] = = (20,10) ⇒ False

{10,20} = = {20,10} ⇒ True

{10,20} = = {10,20} ⇒ True

{10,20} = = {30,10} ⇒ False

{10:20} = = {10:30} ⇒ False

{10:20} = = {20:10} ⇒ False

{10:20} = = {10:20} ⇒ True.

(ii) NOT EQUALTO (!=)

→ In single value data type it will check the waitage between two values

→ It is represented as (!=).

→ If the values are same waitage then it returns False.

→ If the values/waitage is not same then it returns True.

Examples

10 != 20 ⇒ True

10 != 10 ⇒ False

10.05 != 10.05 ⇒ False

10.05 != 10.06 ⇒ True.

10+5j != 10+5j ⇒ False

"abc" != 'abc' ⇒ False

"abc" != "ab" ⇒ True

[10,20] != [10,20] ⇒ False

[10,20] != [10] ⇒ True

(10,20) != (10) ⇒ True

(10,20) != (10,20) ⇒ False.

{10,20} != {10,20} ⇒ False

{10} != {20} ⇒ True

{'a':10} != {'A':10} ⇒ True

{'a':10} != {'a':10} ⇒ False

RELATIONAL OPERATOR :

→ TO (perform) Find out the relationship between two values (or) more than two values.

→ It return boolean values

→ Internally it checks the waitage of values.

Classified into Four(4) types

(i) less than (<)

(ii) lessthan orequal to (<=)

(iii) greaterthan (>)

(iv) greaterthan or equal to (>=)

Note : Not support for complex and dictionary, set not gives perfect result

(i) LESS THAN

→ It will find out relation b/w two values if op1 value is low and operand 2 value is high then it return True

→ If operand 1 value is high than operand 2 then it returns False.

→ If both operand values are same it return False

→ represent as (<)

Example

If case 1 is False it will move on to case 2 or Else case 1 is True it return True

If case 2 is True then it will move no to next Element or Else it return False.

if < if return case 1,

if <= if return case 2

Scenario-1

'ABC' < 'ABC'

65 < 65 case 1 False | 66 < 66 case 1 False | 67 < 67 case 1 False
65 == 65 case 2 True | 66 == 66 case 2 True | 67 == 67 True

If user pass less than it return case(i) result

If user pass less than or equal to it return case(ii) result.

Scenario-2

'ABC' < 'ADC'

65 < 65 case 1 False
65 == 65 case 2 True

66 < 68 case 1 is True
It will stop process and Return case 1 result

Scenario-3

'ADB' < 'BCD'

65 < 66 case 1 True
It will stop process and Return case 1 result

Scenario-4

'ABC' < 'AAC'

65 < 65 case 1 False
65 == 65 case 2 True

66 < 65 case 1 False
66 == 65 case 2 False
It will stop the process and return case 2 result

Ex

[10,20] < [10,20] ⇒ False
[10,20] < [20,10] ⇒ True
[10,20] < (10,20) ⇒ ~~False~~ Error

{10,20} < {20,30} ⇒ False
{10:20} < {20:30} ⇒ Error

(ii) LESS THAN OR EQUAL TO (<=)

→ It will find out relation between two values

If operand 1 value is low and operand 2 val is high then it return True

→ If operand 1 value is high than operand 2 val then it return False

→ If both the operands are equal then it return True

→ Represent as '<=' > lessthan or equal to.

Ex: 65 <= 64 ⇒ False 65 <= 65 ⇒ True

65 <= 66 ⇒ True

(iii) Greater than (>)

→ It will find out relation between two values

If operand 1 value is low and operand 2 value is high then it return False

→ If operand 1 value is high and operand 2 value is low then returns True

→ If both the operands are equal then it return False

→ represent as '>'

Ex: 65 > 64 ⇒ True

65 > 66 ⇒ False

65 > 65 ⇒ False

GREATER THAN OR EQUAL TO (>=)

→ It will find out the relation between two values

→ If operand 1 value is high and operand 2 value is low then it return True

→ If operand 1 value is low and operand 1 value is high then it return False

→ If both the operators are equal then it returns True

Examples

65>=64 ⇒ True

65>=66 ⇒ False

65>=65 ⇒ True

30/01/23
Monday

MEMBERSHIP OPERATOR :

→ It is a searching operator
→ To find out the operand 1 element is present inside the operand 2 or not
→ If element is present it return True.
→ If element is not present it return False.
→ It return Boolean values.
→ The operand 1 value should be any type of value but operand 2 value should be collection type.
→ It is classified into 2 types
  →(i) In operator.
  (ii) not in operator

(i) IN OPERATOR

→ It will find out operand 1 value is present inside the operand 2 or not.
→ If satisfy it return True
→ If not satisfy it return False.
→ Represent as "in". It is a keyword as well as operator.

Syntax

Op_1 in Op_2

Op_1 ↓ Any kind of value

Op_2 ↓ Collection (str, list, tuple, set, dictionary)

Example

→ 'p' in "pys" ⇒ True

→ "py" in "pys" ⇒ True

→ "ps" in "pys" ⇒ False (It is not as it is)

→ 10 in [10,20,30,100]

10 == 10 ⇒ True

→ 100 in [10,20,30,100]

100 == 10 ⇒ False.
100 == 20 → False
100 == 30 ⇒ False
100 == 100 ⇒ True → It stop process and return True.

→ (10) in [10,20,30,100] ⇒ False

→ {10} in {10,50,60,40, "pys", (10,20)} ⇒ False.

→ 10 in {10,50,60,40, "pys", (10,20)} ⇒ True

→ 'pys' in {10,50,60,40, "pys", (10,20)} ⇒ True.

→ (10,) in {10,50,60,40, "pys", (10,20)} ⇒ False

→ (10,20) in {10,50,60,40, "pys", (10,20)} ⇒ True.

→ 'A' in {'A': 10, 'B': "pys"} ⇒ True

→ 10 in {'A': 10, 'B': "pys"} ⇒ False.

In dict it only show keys and for values it return false

→ b'10' in b'1010' ⇒ True.

In operator not support Int, Float, boolean values.

NOT IN OPERATOR

→ To Find out operand 1 value is present inside operand 2 value then it returns False

→ If op_1 value is not present in op_2 then it returns True

→ We Represent not in

not is one key word, in is also a key word as well as Both are operators

Examples

→ 'pys' in 'pyspiders' ⇒ True

→ 'ps' in 'pyspiders' → False

→ 'ps' not in 'pyspiders' ⇒ True.

→ 'py' not in 'pyspiders' ⇒ False.

→ 10 in [10,20,30] ⇒ True.

→ 10 not in [10,20,30] ⇒ False

a = {'a':10, 'b':20}

'e' in 'a' ⇒ False

'c' not in a ⇒ True

BITWISE OPERATOR

→ It is used to perform the operations on Binary values.

→ It is used to perform the operations on individual Bit of the value. and internally in between the two Bits it will perform the logical operations.

→ Here we have to pass/assign integer decimal values

→ Internally the interpreter will take integer values and those values are converted into Binary values and perform the logical operator in between binary values.

→ Final result is converted into decimal value (Binary value ↔ Decimal value).

→ Bit-wise operator is classified into 6 types

(i) Bit wise AND operator

(ii) Bit wise or operator

(iii) Bit wise xor operator

(iv) Bit wise not Operator

(v) Bit wise right shift operator

(vi) Bit wise left shift operator

→ It returns actual values (Integers).

→ Used for Criptography, compressing the data and gaming

(i) BITWISE AND OPERATOR :

→ It will perform the logical and operation in between binary values

→ It return Binary value

→ we represent as '&' operator

Syntax

OP_1 & OP_2

20 & 40 ① Decimal value convert into binary values

2^6 2^5 2^4 2^3 2^2 2^1 2^0

64 32 16 8 4 2 1

0 0 1 0 1 0 0

0 1 0 1 0 0 0

② logical operators based on operator

0 0 1 0 1 0 0

0 1 0 1 0 0 0

0 0 0 0 0 0 0 → 0

③ The result Binary values is converted into decimal values

0 0 0 0 0 0 0

2^6 2^5 2^4 2^3 2^2 2^1 2^0

⇒ 0

19 & 35

2^6 2^5 2^4 2^3 2^2 2^1 2^0

64 32 16 8 4 2 1

19 → 0 0 1 0 0 1 1

35 → 0 1 0 0 0 1 1

→ logical operator based on operator

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

2^1 2^0

→ $2^0 + 2^1 \Rightarrow 1 + 2 \Rightarrow 3$

→ 28 & 17

```
        64 32 16 8 4 2 1
28 →    0  0  1  1 1 0 0
17 →    0  0  1  0 0 0 1
-----------------------
        0  0  1  0 0 0 0
```

⇒ 16

→ 25 & 17

```
        64 32 16 8 4 2 1
25 →    0  0  1  1 0 0 1
17 →    0  0  1  0 0 0 1
-----------------------
        0  0  1  0 0 0 1
```

16 + 1 ⇒ 17

(ii) BITWISE OR OPERATOR:

→ It will perform the logical or operation in between binary values

logical operator

→ It return Binary values

→ we represent as '|'

Syntax

$OP_1 \mid OP_2$

→ 20 | 40

	64	32	16	8	4	2	1
20 →	0	0	1	0	1	0	0
40 →	0	1	0	1	0	0	0
	0	1	1	1	1	0	0

32 + 16 + 8 + 4 = 60

→ 18 | 60

	64	32	16	8	4	2	1
18	0	0	1	0	0	1	0
60	0	1	1	1	1	0	0
	0	1	1	1	1	1	0

32 + 16 + 8 + 4 + 2 ⇒ 62

→ 87 | 6

	64	32	16	8	4	2	1
87 →	1	0	1	0	1	1	1
6 →	0	0	0	0	1	1	0
	1	0	1	0	1	1	1

64 + 16 + 4 + 2 + 1 ⇒ 87

(iii) BIT WISE XOR OPERATOR :

→ It will perfor the logical xor operation in between two binary operations

→ logical operator returns the binary value

→ Represent as '^'

LOGIC OF XOR operation

→ when ever two operant values (binary values) are same it return '0' (zero)

→ when ever two values are there. If one value is different from other then it return '1'.

0	0	0
0	1	1
1	0	1
1	1	0

20 ^ 40

64	32	16	8	4	2	1
0	0	1	0	1	0	0
0	1	0	1	0	0	0
0	1	1	1	1	0	0

32 + 16 + 8 + 4 ⇒ 60

46 ^ 38

64	32	16	8	4	2	1
0	1	0	1	1	1	0
0	0	1	0	0	1	0
0	0	0	1	0	0	0

⇒ 8

iv) BIT WISE NOT OPERATOR

→ It will perform the logical not operator in between the binary value (only one value)

→ Bit wise not operator formula:

$$-(n+1)$$

SYNTAX

~ operand.

→ Represent as negation (~)

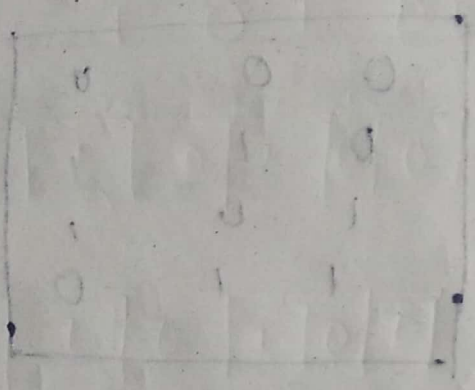
Examples

→ ~20

$-(20+1)$

$-(21)$

$\Rightarrow -21$



→ ~(-21)

$-(-21+1)$

$-(-20)$

$\Rightarrow 20$

BIT WISE RIGHT SHIFT OPERATOR :

→ It will perform eleminating the right side bits based on skipping value and after remaining bits are shifting to right side.

Syntax

$OP_1 >> OP_2$

actual value — skipping value

Example 20 >> 2

	64	32	16	8	4	2	1
20 →			1	0	1	0	0

0	0	1	0	1	0	0

⇒ elimination

0	0	0	0	1	0	1

⇒ right shift

2^6 2^5 2^4 2^3 2^2 2^1 2^0

4 + 1 ⇒ 5

20 >> 2 ⇒ 5

FORMULA : n//(2**s)

n → actual value

s → skipping value

Ex: 20 >> 2

⇒ 20//(2**2)

⇒ 5

→ 20 >> 5

= 20//(2**5)

⇒ 0

BITWISE LEFT SHIFT OPERATOR

vi)

→ It will perform the eliminating the left side bits based on the skipping value and after remaining bits are shifted to left side.

→ Represent <<

Syntax

$Op_1 << Op_2$

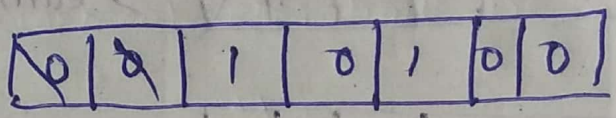
Op_1 → actual value

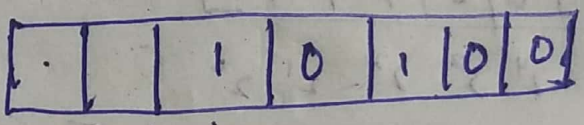
Op_2 → skipping value

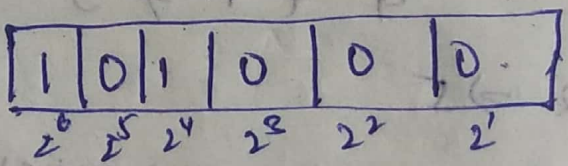
Ex: 20 << 2

64	32	16	8	4	2	1
0	0	1	0	1	0	0

Shifting value = 2







64 + 16 ⇒ 80

FORMULA :- $n*(2**s)$

n → actual value

s → skipping value

20 << 2 → $20*(2**2)$ ⇒ 80

20 << 4 → $20*(2**4)$ ⇒ 320.

31/01/23
Tuesday

ASSIGNMENT OPERATOR :

→ It is a phenominal of assigning the value to the variable name (Identifier).

→ Assigning the memory location adress to the variable name.

→ Represent as '=' (equal to)

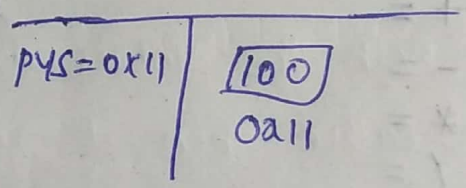
SYNTAX

Variable name = value / New value.

PASS BY VALUE

→ Directly assigning the actual value to the variable name

PYS = 100

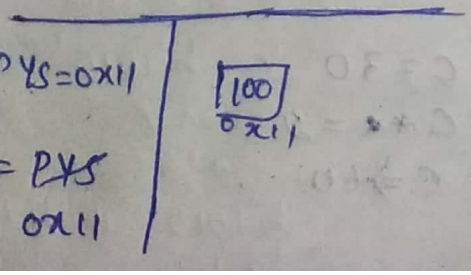


PASS BY REFERENCE

→ Indirectly assigning the value to the variable by the help of existing variable name.

→ Directly passing to the Memory location adress to the new variable name by the help of existing variable name.

QSP = PYS



ADVANCE ASSIGNMENT OPERATOR:

→ REAssigning the resultant value to the existing variable name

→ We represent as combination of two assignment operators.

→ Either arthematical operator or Bit wise operator with assignment operator.

→ This kind of operator we will called as Compound assignment operator

Example

a += 30

Some advance assignment operations are

+=	**=
-=	&=
*=	|=
/=	>>=
//=	<<=
%=	^=

Examples

Normal	Compound
→ a = 10	a = 10
a = a + 10	a += 20
a ⇒ 20	a ⇒ 30
→ b = 20	b = 20
b = b - 10	b -= 10
b ⇒ 10	b ⇒ 10
→ c = 30	c = 30
c = c * 2	c *= 2
c ⇒ 60	c ⇒ 60

Normal

→ d = 10
d = d/2
d ⇒ 5.0

→ e = 20
e = e//2
e ⇒ 10

→ f = 10
f = f%2
f ⇒ 0

→ g = 10
g = g**2
g ⇒ 100

→ h = 20
h = h & 40
h ⇒ 0

→ i = 20
i = i | 40
i ⇒ 60

→ j = 20
j = j ^ 40
j ⇒ 60

→ e = 60
e = e >> 2
e ⇒ 15

→ f = 50
f = f << 2
f ⇒ 200

Compound

d = 10
d /= 2
d ⇒ 5.0

e = 20
e //= 2
e ⇒ 10.

f = 10
f %= 2
f ⇒ 0.

g = 10
g **= 2
g ⇒ 100.

h = 20
h |= 40
h ⇒ 0

i = 20
i |= 40
i ⇒ 60

j = 20
j ^= 40
j ⇒ 60

e = 60
e >>= 2
e ⇒ 15

f = 50
f <<= 2
f ⇒ 200.

IDENTITY OPERATOR

→ TO Identify both the values are pointing to the same memory location or not.

→ It returns bodean values.

→ It is classified into two types

(i) Is operator

(ii) is not operator.

(i) IS OPERATOR

→ To Identify both the values are pointing to same memory location or not.

→ If both values are pointing to same location it return True.

→ Both the values are not pointing to same location then it return False

Examples

→ a = 10
b = 10
a is b
→ True

→ i = 10.05
j = 10.05
i is j
False

→ m = 10+5j
n = 10+5j
m is n
False
10+5j is 10+5j
True

→ p = True
0 = True
p is 0
True

→ i = None
j = None
i is j
True

→ u = b'1010'
v = b'1010'
u is v
False
b'1010' is b'1010'
True

→ a = 'hello'
b = 'hello'
a is b
True

→ a = '1hello'
b = '1hello'
a is b
True.

→ a = 'hello*haii'
b = 'hello *haii'
a is b
False

→ If any symbol is available inside the String every time it will generate the new reference id for the string value.

→ If symbol is not available inside the String then it will consider old reference value.

(ii) IS NOT OPERATOR

→ To Identify the both the values are pointing to same memory location or not.

→ If both values are pointing to same memory location then it return False.

→ Both the values are pointing to different memory location then it return True.

Ex: a = [10, 20, [10, 20], "haii"]
b = [10, 20, [10, 20], "haii"]

→ a is b
False

→ a[0] is b[0]
True

a[2] is b[2] ⇒ False

a[2][0] is b[2][0] ⇒ True.

a[-1][0] is b[-1][0] ⇒ True

ASSIGNMENT CONCATINATION OPERATION

Operator is (+=)

SYNTAX

String

a = " "

a = a + value / value should

a += value → value should be string

List

a = [ ] Internally ⇒ a += vale
a += list(a)

a += value → value of any type (str, list, tuple, set, dict)

tuple

a = ( )

a += value → value should be tuple

Example

a = [ ]

a += "str"

⇒ a = ['s', 't', 'r']

a += [10, 20]

⇒ a = ['s', 't', 'r', 10, 20]

a += (30, 40)

⇒ a = ['s', 't', 'r', 10, 20, 30, 40]

a += {50, 60}

⇒ a = ['s', 't', 'r', 10, 20, 30, 40, 50, 60]

a += {'a': 70, 'b': 80}

⇒ a = ['s', 't', 'r', 10, 20, 30, 40, 50, 60, 'a', 'b']

→ b = "srr"

b+ = "hai"

b = "srrhai"

→ C = ()

C += (150, 160, 180)

C = (150, 160, 180)

Saturday 4/2/23

CHR and ORD

Ord function

→ It is a utility pre define function. This function will convert single character to ASCII value

→ It returns ASCII value in integer format

→ SYNTAX:

ord(character)

↓

Must be single character

Chr function:

→ It is a predefine utility function. This function will convert ASCII value to single ASCII character.

→ It returns ASCII character

Syntax

Chr(Integer number)

↓

(ASCII value)

11/12/23 Monday

Input and output statements:

INPUT

→ Assigning the values to the variable name.
→ Classified into two types.
  → Static input Assignment (Static initilization)
  → Dynamic input Assignment (Dynamic initilization)

Static initilization

→ when writing the program directly passing the values to the variable names.
→ Assigning values to the variable names before running the program

Example

Shell file

```
a=10
b=[10,20,30]
c= "Hello".
```

Dynamic initilization:

→ when the program is executing in the running state, that time if you passing the values/ assigning the values this process we called as Dynamic initilization.
→ Assigning the values to the variable names after running the program.
→ If we want to assigning/pass the value in running time we have to use one buitin function name called input function.

input():

→ It is a predefine utility function
→ This function is execute in program Execution/running state.
→ It will ask the input from the shell. in running state / program execution time.
→ It will take the input from the user, that input stores in the form of "String" format
→ It returns string format input to the variable.

Swamy

SYNTAX

varname = input("message")

NOTE :- Message is an optional

Examples

→ a = input("Enter the integer value :")
enter the integer value :10
a = '10'

→ b = input("enter the complex value :")
Enter the complex value : 10+5j
b = '10+5j'

→ c = input("enter the string value :")
Enter the string value : "hello"
c = '"hello"'

→ d = input("Enter the string value :")
Enter the string value : hello
d = 'hello'

→ e = input("enter the list value :")
Enter the list valu : [10,20,30]
e = "[10,20,30]"

Converting string to another/other data type

→ a = "10"
int(a) ⇒ 10

→ float('10.05') ⇒ 10.05

→ Complex('10+5j') ⇒ 10+5j

→ bytes("b'1010'") ⇒ Error (not possible)

→ list['[10,20,30]'] ⇒ ['[', '1', '0', ',', '2', '0', ',', '3', '0', ']']

→ tuple('(10,20,30)')

('(', '1', '0', ',', '2', '0', ',', '3', '0', ')')

→ set('{10,20,30}')

⇒ {'2', '}', '3', ',', '{', '0', '1'}

→ dict("{10:20}") ⇒ Error (not possible).

---

converting string to other data types using type casting in dynamical input

→ a = int(input("enter the integer value:"))

enter the integer value: 12

a ⇒ 12

→ a = float(input("enter the float value:"))

enter the float value: 12.06

a ⇒ 12.06

→ a = complex(input("enter the complex value:"))

enter the complex value: 10+5j

a ⇒ 10+5j

→ a = bool(input("enter the bool value:"))

enter the bool value: 0

→ a = list (input("enter the list value :"))
enter the list value : [10,20,30]
a ⇒ ['[','1','0',',','2','0',',','3','0',']']

→ a = tuple (input("enter the tuple value :"))
enter the tuple value :aa
a ⇒ ('a','a').

→ a = set (input("enter the set value :"))
enter the set value : {10,20}.
a ⇒ {'2','}',',','{','0','1'}

Note:
⇒ For list, set, tuple, string we have some problems so to overcome that problem we use 'eval function

→ b = eval (input ("enter the value :"))
enter the value : 12
b ⇒ 12

→ b = eval (input("enter the value :"))
enter the value : "string"
b ⇒ string

→ b = eval (input("enter the value :"))
enter the value : [10,20,30]
b ⇒ [10,20,30]

→ b = eval (input("enter the value :"))
enter the value : ~~{10,20,30}~~ {a:10}
b ⇒ {a:10}

Here eval() returns the value/datatype as the input datatype.

eval ( )

→ It is a predefined utility function.

→ It will take the string format input and it returns actual data type

→ It is a evalute function

→ It is evaluting the string format values to actual values.

SYNTAX

eval ("string input") (or) eval ("value")

Examples

→ eval ("10") ⇒ 10

→ eval ("10.05") ⇒ 10.05

→ eval ("10+5j") ⇒ 10+5j

→ eval ("b'1010'") ⇒ b'1010'

→ eval ('True') ⇒ True

→ eval ("'str'") ⇒ str

→ eval ('[10,20,30]') ⇒ [10,20,30]

→ eval ('(10,20)') ⇒ (10,20)

→ eval ('{10,20}') ⇒ {10,20}

→ eval ('{10:20}') ⇒ {10:20}