**Day-01**

**07-03-2025**

**=================**

Operand:

======

-> operand is a value on which we can define any operation

Ex: 10 - 5

here:

 - ==> operation

 data: 10 and 5 ==> operands

-> for operand, we can use a variable also.

Ex: x = 100

y = 2

x\*y ==> 200

Operator:

=======

-> Operator is a symbol

that we can use to denote the operation

from above examples:

 - ==> subtraction

 \* ==> multiplication

-> Operator is also a keyword

Ex: x = 10

y = 20

x < y and x > y

Expression

=======

-> Expression is the combination of operators and operands

Ex: x - y

x \* y

x < y and x > y

Statement

=======

-> The executable part of the program is called as "statement".

-> Every line of the program is a statement.

-> The statement instructs what to do.

-> In other programming languages like Java, c++, c the statement must be ended or terminated with semi-colon but in python the semi-colon is optional in statement.

Block

-----

-> block is a group of statement

-> block can be defined with "indentation"

Ex:

if True:

 print("Hi")

 print("Good morning")

 print("Welcome To Ashok IT")

Type of Operators:

=============

-> According to the operands, the operators are classified into three types:

 1) Unary Operators

 2) Binary Operators

 3) Ternary Operators

-> When any operator can define with single operand such type of operators are called as "Unary operators".

Ex: number = 10

the above number is positive

to make negative:

 number = -10

for the changing of the sign:

 we have used "unary minus"

-> Binary operators are always use with two operands.

Ex: 10 + 20

-> Ternary operator can define with more than two operands

Ex: Conditional operator

-> The operators are classified into several based on the operations:

 1) Arithmetic operators

 2) Assignment Operators

 3) Logical Operators

 4) Relational Operators

 5) Bitwise Operators

 6) Conditional Operators

 7) Special Operators

1) Arithmetic operators

---------------------------

 + (sum), - (subtraction), \* (multiplication),

 / (normal division), // (floor division), % (Modulo Division) ==> division operators

 \*\* (Exponet Operator/Power Operator)

a = 10

b = 20

print(a + b)

print(a - b)

print(a \* b)

# normal division ==> can give a quotient in float format

print(a/3)

# floor division ==> can give quotient in integer format

print(b//3)

# modulo division ==> can give a remainder

print(a % 3)

print(b % 3)

print(a \*\* 4)

print(a \*\* b)

2) Assignment Operators

-----------------------------

x = 10 # variable assignment

10 (right side) can assign to the variable-name 'x'

compound operators:

 +=, -=, \*=, /=, //= , %= , \*\*= etc.

x = 10

print(x)

# x = x + 10

x += 10

print(x)

# x = x \* 3

x \*= 3

print(x)

3) Relational Operators

---------------------------

-> Also called as "Comparison Operators".

-> The relational operators are:

 <, >, <=, >=

 Equal operator (==)

 Not Equal operator (!=)

-> The output of these operators are:

 boolean values

**Q: What is the observation when we can use '=' in the place of '==' and '==' in the place of '='?**

Ans:

If we can use '=' in the place of '==' ==> "Type Error"

If we can use '==' in the place of '=' ==> Name Error

a = 10

b = -10

print(a < b)

print(a > b)

print(a <= b) # 10 < -10 or 10 == -10

print(a >= b) # 10 > -10 or 10 == -10

print(a == b)

print(a != b)

# print(a = b)

4) Conditional Operators

----------------------------

-> Also called as "Ternary Operator"

Syntax:

 Expression-1 if condition else Expression-2

Here:

 condition can always define with relational operators

 the value of the condition is "boolean".

The program controller can check or evaluate the condition

if the condition is "True", "Expression-1" can execute.

otherwise (condition is "False" ) "Expression-2" can execute.

**# WRITE A PYTHON PROGRAM USING CONDITIONAL OPERATOR TO FIND WHETHER THE**

**# GIVEN NUMBER IS POSITIVE OR NEGATIVE.**

**# positive number is always greater than to 0**

**# negative number is always less than to 0**

number = int(input("Enter a value:"))

print("The given number is positive") if number > 0 else print("The given number is negative")

**# WRITE A PYTHON PROGRAM TO ACCEPT THE USER AGE AS AN INPUT**

**# CHECK WHETHER THE USER IS ELIGIBLE TO VOTE OR NOT.**

age = int(input("Enter the user's age:"))

print("The person is eligible to vote") if age >= 18 else print("The person is not eligible to vote")

**# WRITE A PYTHON PROGRAM TO ACCEPT A NUMBER AS AN INPUT.**

**# CHECK THAT NUMBER IS EVEN OR ODD.**

**# Even number ==> when a number is evenly divided with 2**

**# Odd number ==> when a number is not evenly divided with 2**

number = int(input("Enter a value:"))

remainder = number % 2

print("The given number is Even number") if remainder == 0 else print("The given number is Odd number")

Assignment:

========

1) WRITE A PYTHON PROGRAM TO FIND WHICH NUMBER IS BIGGER AMONG THE TWO GIVEN INTEGERS USING CONDITIONAL OPERATOR.

**Day-02**

**08-03-2025**

**==================**

5) Logical Operators:

------------------------

-> There are three logical operators:

 1) Logical and --> and

 2) Logical or --> or

 3) Logical not --> not

-> Among these three,

 logical and & logical or are binary operators

 whereas the logical not is the unary operator.

-> Logical operators allowed to define with any primitive type of value.

Because, PVM can internally understand:

 any non-zero value as "True"

 and zero as "False".

Logical and:

--------------

1) When both inputs are "True" then: the output is "True"

2) If any of the input is "False" then, the output is "False".

-> When the first input is "True" (any non-zero value) then the output of the logical and is "second input value".

-> When the first input is "False" (any zero value) then the output of the logical and is "first input value".

# with boolean values

print(True and True),print(True and False),print(False and True),print(False and False)

# with integers

print(1 and 1) # both are non-zeros 1

print(1 and 0) # second one is zero 0

print(0 and 1) # first one zero 0

print(0 and 0) # both are zero 0

print(-1 and -11) # both are negative values -11

# with floats

print(1.23 and 0.0)

print(0.0001 and 0.010101)

print(0.00000000 and 0.00000000010101)

# with complex numbers

print(1+0j and 0-1j)

print(0j and 1-1j)

# with strings

print('a' and 'b')

print(' ' and 'r')

print('' and 'p')

Logical or

-----------

1) When any input is "True" then, the output of the logical or is "True"

2) When both inputs are "False" then, the output of the logical or is "False".

-> if the first operand is "True" (non-zero value) then, the output is "first-operand-value"

-> if the first operand is "False" (zero value) then, the output is "second-operand-value".

# with boolean values

print(True or True)

print(True or False)

print(False or True)

print(False or False)

# with integers

print(1 or 1) # 1

print(1 or 0) # 1

print(0 or 1) # 1

print(0 or 0) # 0

print(-1 or -10) # -1

# with strings

print('a' or 'b'),print(' ' or 'r')

print('' or 'n')

Note:

-----

1) If we can define logical and or logical or operators with boolean values, the output format is also "boolean".

2) If we can define with other than boolean:

 inputs are with integers ==> output should be "integer"

 inputs are float ==> output should be "float"

Logical not

-------------

-> For any type of input, the output format is always in "boolean".

-> alters the input

print(not True)

print(not False)

print(not 100)

print(not 0)

print(not 0-1j)

print(not 1.001)

print(not 0.000)

print(not 'a')

print(not '')

**Day-03**

**10-03-2025**

**================**

Bitwise Operators:

---------------------

bits ==> 1 and 0

binary value ==> 0b1010101

Ex: data = 10

convert data to binary

10 ==> 1010

Note:

-----

Bitwise operations are not allowed to define with float data, complex data and string data.

-> The bitwise operators are:

 1) Bitwise and ==> &

 2) Bitwise or ==> |

 3) Bitwise xor ==> ^

 4) Bitwise not ==> ~

 5) Left shift ==> <<

 6) Right shift ==> >>

1) Bitwise and ==> &

-------------------------

-> binary operator

Syntax:

 operand1 & operand2

-> operand1 and operand2 can convert into binary

after the conversion, from right to left bit by bit the bitwise and operation can be performed.

-> truth table:

 a b a & b

 ----------------------

 0 0 0

 0 1 0

 1 0 0

 1 1 1

when both operands are with '1' then, the result = '1'

if any operand is '0' then, result is = '0'





2) Bitwise or ==> |

-------------------------

-> binary operator

Syntax:

 operand1 | operand2

Ex: 10 | 20

-> truth table

 a b a | b

 ---------------------

 0 0 0

 0 1 1

 1 0 1

 1 1 1

when both inputs are '0' then only the result is '0'

in all the remaining cases the output should be '1'

-> operand1 and operand2 should convert into binary

after the conversion, the operation can implement from right to left bit by bit.



3) Bitwise xor ==> ^

--------------------------

-> binary operator

Syntax:

 operand1 ^ operand2

Ex: 10 ^ 12

here:

10 and 12 ==> convert into binary

after that, from right to left bit by bit xor operation can define.

-> truth table:

 a b a ^ b

 ===========

 0 0 0

 0 1 1

 1 0 1

 1 1 0

-> when operand1 and operand2 with same value then, the result = '0'

and both operand's values are different/alter then, the result = '1'

print(10 & 20)

print(0o12 & 0x12)

print(10 | 20)

print(0o12 | 0x12)

print(10 ^ 12)

# print(1.2 & 2.3) Type Error

# print(1.2 | 2.3)

# print(1.2 ^ 2.3)

# print(1+2j & 2-3j)

# print('a' | 'b')

print(True & True)

print(True | False)

print(True ^ False)



4) Bitwise not ==> ~

-------------------------

-> also called as "bitwise complement" operator.

-> unary operator

-> can invert the data

 + eve --> -eve

 - eve --> +eve

-> The data can convert into binary

that binary may have sign bit

here:

 0 ==> +

 1 ==> -

-> 2's complement ==> 1's complement + 1

-> 1's complement ==> in the binary data, 0's should convert into 1's and 1's can into 0's.

-> while the 1's complement calculation, keep the sign bit as same (not to alter the sign bit).

-> after the calculation of 1's complement we need to add 1 (0001) to the result.

-> If there is a carry then, we need to rounding-off that carry means the carry bit can add to the last bit of the data and change the sign bit.



-> If there is no carry, we need to repeat the same operation on the result what we have achieved. While this operation, the sign bit can change first.



-> Negative numbers:



print(~0)

print(~1)

print(~-1)

**Day-04**

**11-03-2025**

**==================**

Left shift Operator (<<)

--------------------------

-> Binary operator

Syntax:

 data/operand1 << n/operand2

here:

 data --> on which we need to define left shift

 n --> specifies the number of times

Ex: 12 << 2

left shift on 12 for 2-times

-> The left shift operator can convert operand1/data into binary

then perform the left shift operation.

-> The left shift operation describes,

after the data converted into binary, from right to left the bits of the data shift bit by bit. After the shifting of all the bits, there is the vacancy at right most bit (LSB == Least Significant Bit). That corresponding vacant can fill with '0'.

-> Because of the left shift operation, the value/data can be doubled for every time.

-> Math formula for left shift operation:

 data X 2^n

Ex: data = 12

n = 2

left shift result = 12 X 2^2

==> 12 X 4 ==> 48



print(12 << 2)

print(0o21 << 3)

print(0x21 << 3)



Right shift operator (>>)

--------------------------

-> Binary operator

Syntax:

 data >> n

Ex: 12 >> 2

right shift on 12 for 2 times

-> During the right shift operation:

i) data can convert into binary

ii) after the conversion, the bits can shift from right to left bit by bit.

iii) during the shift operation, the LSB old bit can discard

iv) and the vacant at Left most bit (MSB == Most Significant Bit)can fill with '0'

print(12 >> 2)

print(0o21 >> 2)

print(0x21 >> 3)



**Special Operators:**

**============**

1) Identity Operators/Address operators

----------------------------------------------

-> can check whether the object's address is same or not.

id():

-----

 can give the address of any data.

 Syntax:

 id(data/variable)

a = 121

b = 1.21

print("id(a) = ",id(a))

print("id(b) = ",id(b))

-> two types of identity operators:

 1) is

 2) is not

-> return boolean values as an output.

is:

 True -> when two object's address is same

 False -> when two object's address is different

is not:

 True -> when two object's address is different

 False --> when to object's address is same

a = 10

b = 20

c = a

print("address of a = ",id(a))

print("address of b = ",id(b))

print("address of c = ",id(c))

print(a is b)

print(a is c)

print(a is not b)

print(a is not c)

2) Membership Operators

------------------------------

-> to check whether the element is belonging to the collection/string or not.

-> two membership operators:

 1) in

 2) not in

-> these are also return boolean values

in:

 True -> when the specified element is belonging to the main data

 False -> otherwise

not in:

 opposite to in operator

string = "Python"

print('t' in string)

print('t' not in string)