**Operators**

**Day-01**

**27-05-2025**

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

**Arithmetic and Logical Statement:**

Operand

 Ex: 10 + 20

-> The data on which we can define the operation to perform called as “operand”.

From the above example:

 10 and 20 🡺 operands

Expression

-> The combination of operands and operators is called as an “Expression”.

Ex: 10 + 20 \* 3 / 4

Here:

 10, 20, 3 and 4 🡺 operands

 +, \*, / 🡺 describe the operation

Statement

-> Every line of the program is called as “statement”.

-> Other high-level programming language can end/terminate with semi-colon.

-> In python semi-colon (😉 is optional while defining the statement.

Block

-> Group

-> The group of statements called as “block”.

-> The block of code in python can able to define with indentation (1 tab space 🡺 4-sequel spaces).

Ex:

if True:

 print(“Hi”)

 print(“Good Evening”)

 print(“We are starting to learn about Arithmetic and logical statement”)

Q-1: Do we declare variables in python?

Ans: No, Because the python is “Dynamically typed programming language”. This is the reason we are not using the datatype before the variable name as like other programming languages. Without datatype, declaration is not fulfill. So, there is no declaration and initialization for the variable. We just have only the variable assignment in python.

a = 123 # assignment

print(a)

Q-2: Do we define more than variable within the same line?

Ans: Yes

name = “Karthik”;age = 22;location = “Hyderabad”

print(name)

print(age)

print(location)

# using comma

name, age, location = “Kishore”, 24, “Mumbai”

print(name)

print(age)

print(location)

In Java:

 variable declaration:

 datatype variable;

 Ex: int a;

Variable:

-> can be defined as a name

 can be used to store the value of any type.

Note:

1) When we can define a variable, the PVM can create the new memory location/block for the value in heap memory and for the name (variable) the memory block can be created in stack memory.

2) Once the memory blocks can be created, from stack memory block to heap memory block there is dynamic linking can be created.

3) When we can re-assign the same variable with new value,

 The stack memory block for the variable remains same but the previous the dynamically linked heap memory block can be de-linked and deleted that value permanently. And make create the new memory block for the new value in the heap and make the dynamic linking from that variable’s static block to the new heap memory block.

This can be happened for the variable which are defined with “Immutable” type of data.

-> Immutable datatypes: The data cannot be modified/changed within the same memory location for every change the new memory location can be trigger. Such datatypes are called as “Immutable datatypes”.

Ex: Primitive datatypes

-> Mutable datatypes: can allow to modify/change the value within the same memory location again and again.

Ex: List, Set, Dictionary etc.

Q: Is it possible to define more than one statement in the same line.

Ans: Yes

print(“Hi”),print(“Good Evening”),print(“Welcome To Ashok IT!”)

print(“Hi”);print(“Good Evening”);print(“Welcome To Ashok IT!”)

**Operators:**

-> Operator is a symbol

can be used to define/specify the operation

Ex: 100 – 200

here:

 100, 200 🡺 operands

 - 🡺 subtraction

-> The operators are classified into three types, based on the number of operands:

 1) Unary Operators

 2) Binary Operators

 3) Ternary Operators

-> Unary operators can allow to define with single operand.

Ex: -7 -> 7 is negative

-> Binary operators can allow to define with two operands.

Ex: 10 + 20

-> Ternary Operator can allow to define with more than two operands.

Ex: Conditional operator

-> Based on the operations, the operators are classified into:

 1) Arithmetic Operators

 2) Assignment Operator

 3) Relational Operators

 4) Logical Operators

 5) Conditional Operator

 6) Bitwise Operators

 7) Special Operators

**Day-02**

**28-02-2025**

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

**Arithmetic Operators:**

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

 + ==> plus ==> addition ==> binary

 - ==> minus ==> subtraction ==> binary

 \* ==> asterisk ==> multiplication => binary

 / ==> slash ==> normal division => binary

 can return the quotient in float

 // ==> double slash ==> Floor division ==> binary

 can return the quotient in integral format

 % ==> percentage ==> Modulo division ==> binary

 can return a remainder value

 \*\* ==> double asterisk ==> Exponent/Power ==> binary

 unary + ==> to specify the number is positive ==> unary

 Ex: +9

 unary - ==> to specify the number is negative ==> unary

 Ex: -9

12 + 24 ==> 36

24 - 12 ==> 12

6 \* 5 ==> 30

9 / 4 ==> 2.25

9 // 4 ==> 2

9 % 4 ==> 1

3 \*\* 4 ==> 3^4 ==> 81

**# WAP TO ACCEPT THE PRICE OF TWO PRODUCTS AND DISPLAY THE TOTAL PRICE OF ALL PRODUCTS IN A CART.**

product1 = float(input("Enter the price of the given product:"))

product2 = float(input("Enter the price of the given product:"))

total = product1 + product2

print("The Total cost of the cart = ",total)

**# WAP TO ACCEPT THE AMOUNT THAT USER HOLD. AND ALSO ACCEPT THE COST PRODUCT THAT USER WANT TO BUY.**

**# CALCULATE THE REMAINING AMOUNT AFTER THE USER'S PURCHASE.**

amount = 1999.93

cost = 1099.19

remaining\_balance = amount - cost

print("The Remaining balance after the purchase = ",remaining\_balance)

**# WAP TO ACCEPT THE NUMBER WORKING HOURS OF AN EMPLOYEE. AND CONSIDER Rs. 1000 PER HOUR OF HIS WORK.**

**# CALCULATE THE TOTAL SALARY OF THE EMPLOYEE.**

totalWorkingHours = int(input("Enter the total working hours:"))

salary = totalWorkingHours \* 1000

print("The Salary of the Employee = ",salary)

**# WAP TO CALCULATE THE BILL OF EACH PERSON.**

**# WHEN THE GROUP OF 8 PEOPLE VISITED A RESTAURANT.**

**# THE TOTAL OF BILL OF ALL 8-PEOPLE IS 5034/-.**

totalPersons = 8

totalBill = 5034

costOfPerson = totalBill / totalPersons

print("The Cost of Each person = ",costOfPerson)

**# WAP TO CALCULATE THE TOTAL NUMBER OF CHOCOLATES AT EACH STUDENT BY CONSIDERING THE BELOW:**

**# THERE IS A CLASS OF 20 STUDENTS**

**# THE TOTAL CHOCOLATES THAT TEACHER BRING ARE 47.**

**# DISTRIBUTION OF THE CHOCOLATES AMONG STUDENTS MUST BE SAME**.

totalStudents = 20

totalChocolates = 47

chocolateAtOneStudent = totalChocolates // totalStudents

print("The Total chocolates at each student = ",chocolateAtOneStudent)

**Assignment Operator:**

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

 = ==> we can use to assign the value to variable/expression etc.

Syntax:

 variable = value

-> assignment operator can always work from right to left direction.

-> With the assignment operator, we can define the different operators are called as "compound operators".

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

a = 10

# a = a + 10

a += 10

print(a)

# a = a - 7

a -= 7

print(a)

# a = a \* 3

a \*= 3

print(a)

**Day-03**

**03-03-2025**

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

**Relational Operators:**

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

-> Also called as "Comparison Operators".

-> Return value of relational operators are "Boolean values (True and False)".

-> Relational Operators are:

 < --> Less than

 > --> Greater than

 <= -> Less than or equal

 >= -> Greater than or equal

 == -> Equals to

 != -> Not Equals to

-> All these operators are "Binary operators".

-> The relational operators are allowed to define on any integer value.

when we can define the binary or octal or hexadecimal for the comparison,

here: the binary or octal or hexadecimal can convert into decimal and then do the comparison.

-> Comparison on complex numbers is not allowed.

If we can define, we can get the "Type Error".

-> While defining the string comparison,

 the relational operators can consider the uni code values of every character in string and then compare.

Ex:

'a' < 'A'

97 < 65 -> False

'PyThOn' > 'Python'

'P' > 'P'

# with integers

print(10 < 20)

print(10 > 20)

print(10 <= 20) # 10 < 20 or 10 == 20

print(10 >= 20) # 10 > 20 or 10 == 20

print(10 == 20)

print(10 != 20)

print(0b1001 < 0o102)

# with floats

print(1.002 < 2.3021)

# with complex

# print((2-3j) > (3-2j))

# with boolean

print(True > False)

# with strings

print('a' < 'A')

print('a' > 'A')

print('pyThOn' > 'Python')

print('PYTHON' > 'python')

**Logical Operators:**

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

-> Three logical operators:

 1) Logical and ==> and

 2) Logical or ==> or

 3) Logical not ==> not

-> Logical and, Logical or operators are binary operators

and logical not is unary operator.

->Logical operators works based on the truth table.

-> Logical operators by default can allow to define with "boolean values".

Logical and

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

 a b a and b

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

 True True True

 True False False

 False True False

 False False False

-> When both inputs are "True", then the output is "True"

if any input is "False" then the output is "False".

How the logical and can work internally?

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

1) When the first input (left operand) is "True", then the logical and operator can decode/execute the second operand (right operand). If the second operand is "True" then, the output is "True". Otherwise, the output is "False".

2) When the first input is 'False' then, the logical and operator gives the output as "False" without decoding/executing the second operand.

3) When the first operand is "True" or "other than 0", the output of the logical and is "second operand value".

4) When the first operand is "False" or "Zero-value", the output of the logical and is "first opened" without decoding/executing the second operand.

print(True and True)

print(True and False)

print(False and True)

print(False and False)

print(10 and -12)

print(-10 and True)

print(20 and False)

print(0 and 10)

print(False and -1)

print(12.3 and 0.001)

print(1-2j and 0.0j)

print('a' and 'b')

2) Logical or

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

 a b a or b

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

 True True True

 True False True

 False True True

 False False False

-> When any of the input is "True" then, the output is "True"

When both inputs are "false" then only the output is "False".

How the logical or can internally work?

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

1) When the first operand is "True" or "non-zero" value then, output is "First operand only" (without decoding or executing the second input)

2) When the first operand is "False", or "zero" then, output is "second operand".

print(True or True)

print(True or False)

print(False or True)

print(False or False)

print(True or 121)

print(121 or True)# 121

print(False or 102)

print(0 or False)

3) Logical not:

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

Syntax:

 not operand

 a not a

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

 True False

 False True

-> When we can define the logical not on any value other than boolean, we can get the output as "boolean" only.

print(not True)

print(not False)

print(not 0)

print(not 100)

**Day-04**

**04-03-2025**

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

**Bitwise Operators:**

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

Ex: 10 ==> 0b1010

-> The bitwise operators can perform the operations on the data bit by bit.

-> The bitwise operators are:

 1) bitwise and ==> &

 2) bitwise or ==> |

 3) bitwise xor ==> ^

 4) bitwise complement ==> ~

 5) left shift operator ==> <<

 6) right shift operator ==> >>

Note:

-----

-> Bitwise operators in python can allowed to define on:

 binary data

 octal data

 hexadecimal data

 decimal data

 boolean data

but not allowed to define with:

 float data

 complex data

 string data

-> When we can define the bitwise operations on the numbers, we can get the output in decimal

Whereas when we can define the bitwise operations on boolean type, we can get the output as "boolean".

1) bitwise and ==> &

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

-> Binary operator

Syntax:

 operand1/value1 & operand2/value2

-> Truth table of the bitwise and (&) is:

 a b a & b

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

 0b0 0b0 0b0

 0b0 0b1 0b0

 0b1 0b0 0b0

 0b1 0b1 0b1

-> bitwise and can return the binary '1' (0b1) when both operand values are '0b1'

When any operand is defined with '0b0',the output is '0b0'

-> bitwise and operator can work from left to right for the data conversion

Ex: a & b

a ==> should convert into binary

b ==> should convert into binary

after the conversion of the data into binary, bit by bit the truth table can implement from right to left.

a = 0b1010101

b = 0o124

print(a & b)

c = 10

d = 12

print(c & d)

# print(1.2 & 2.3)

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

# print('a' & 'b')

print(True & False)

print(False & True)

print(False & False)

print(True & True)

2) bitwise or ==> |

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

-> Binary operator

Syntax:

 operand1 | operand2

-> The bitwise or also can depend on the truth table:

 a b a | b

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

 0b0 0b0 0b0

 0b0 0b1 0b1

 0b1 0b0 0b1

 0b1 0b1 0b1

-> When any input/operand is '0b1' the output is '0b1'

if both inputs are '0b0', then only the output is '0b0'.

print(0b0 | 0b0)

print(0b0 | 0b1)

print(0b1 | 0b0)

print(0b1 | 0b1)

print(10 | 12)

print(True | True)

print(True | False)

print(False | True)

print(False | False)

3) bitwise xor ==> ^

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

-> Binary operator

Syntax:

 operand1 ^ operand2

-> The bitwise xor is also working based on the truth table:

 a b a ^ b

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

 0 0 0

 0 1 1

 1 0 1

 1 1 0

-> When both operands are defined with same values, we can get the output as '0'.

For altered inputs, the output should be '1'

print(0 ^ 0)

print(0 ^ 1)

print(1 ^ 0)

print(1 ^ 1)

print(12 ^ 10)

print(True ^ True)

print(True ^ False)

print(False ^ True)

print(False ^ False)

4) bitwise complement ==> ~

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

-> Unary operator

Syntax:

 ~ operand

-> for the given value, the 2's complement is the result.

-> bitwise complement can convert the data from positive to negative or negative to positive.

print(~0) # 0 - 1 ==> -1

print(~10) # -10 - 1 ==> -11

print(~-10) # 10 - 1 ==> 9

print(~True) # -1 -1 ==> -2

**Day-05**

**05-05-2025**

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

**5) Left shift Operator:**

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

Syntax:

 operand1 << operand2

Here:

 operand1 ==> the data on which we can define the left shift

 operand2 ==> the number of times

Ex: 97 << 3

97 can shift from right to left in 3 times

-> The left shift operator can convert the left operand (operand1) into binary

Then from right to left (LSB to MSB) the bits can shift bit by bit and the vacant at LSB fill with '0'

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

-> The math formula for left shift operation is:

 data X 2^n

here:

 n ==> the number of times

print(97 << 1)

print(97 << 2)

print(97 << 3)

**6) Right shift operator:**

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

Syntax:

 operand1 >> operand2

Here:

 operand1 ==> data on which we need to define right-shift

 operand2 ==> number of times

Ex: 9 >> 2

the bits of '9' can shift from left to right bit by bit by 2 times.

-> The right shift operator can convert the operand1 into binary and the bits of the data can shift from left to right bit by bit. When the bit shifting to LSB place, the previous bit at that position (LSB) can be discarded and vacancy at MSB cab fill with '0'.

-> Because of the right shift operation, the data can be halved for every definition.

print(97 >> 1)

print(97 >> 2)

print(97 >> 3)

-> Math formula for right shift is:

 data // 2^n

**Conditional Operator:**

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

Syntax:

 expression-1\_True if condition else expression-2\_False

-> Ternary Operator

-> From the above syntax,

the PVM can check/evaluate the "condition"

if the condition is "True" then, "expression-1" can execute and return a value as an output.

Otherwise, the "expression-2" can execute and return a value as an output.

**# WAP IN PYTHON TO CHECK WHETHER THE GIVEN IS POSITIVE OR NEGATIVE.**

**"""**

**n > 0 ==> positive**

**n < = ==> negative**

**"""**

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

print(n,"is positive") if n > 0 else print(n,"is negative")

**# WAP IN PYTHON TO ACCEPT A NUMBER AS AN INPUT**

**# AND FIND THE GIVEN NUMBER IS EVEN NUMBER OF ODD NUMBER**

**# even number: when a number is evenly divided with '2' (returns a remainder '0') that number is called**

**# as "even number"**

**# odd number: the number which is not evenly divided with '2' is called as "odd number"**

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

print(number,"is even number") if number % 2 == 0 else print(number,"is odd number")

**# WAP IN PYTHON TO ACCEPT TWO INTEGERS AND FIND THE SUM OF THOSE NUMBERS IF**

**# FIRST NUMBER IS LESS THAN SECOND NUMBER**

**# AND FIND THE DIFFERENCE BETWEEN THEN WHEN FIRST NUMBER IS GREATER THAN THE SECOND.**

num1 = int(input("Enter the first value:"))

num2 = int(input("Enter the second value:"))

result = (num1 + num2) if num1 < num2 else (num1 - num2)

print("The Result after the defined operation = ",result)

**Special Operators:**

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

1) Identity Operators

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

-> also called as "Address operators".

-> Binary operator

-> two types address operators:

 1) is

 2) is not

-> the identity operators can always return boolean value as an output by comparing the addresses of two operands.

If two operand's addresses are same:

 is ==> True

 is not ==> False

If two operand's addresses are different:

 is ==> False

 is not ==> True

id()

====

-> can use to get the address of the any data

Syntax:

 id(data/variable)

a = 10

b = 10

c = 20

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 not b)

print(a is c)

print(a is not c)

2) Membership Operators

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

-> Two membership operators:

 1) in

 2) not in

-> Membership operators can use on:

 strings

 and other reference type of the data

-> Membership operators can check the given element/value is present in the given string/other collections or not.

If present:

 in ==> True

 not in ==> False

If not present:

 in ==> False

 not in ==> True

s = "Python"

print('t' in s)

print('t' not in s)

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

print(30 in l)

print(30 not in l)

Assignment:

===========

1) WAP IN PYTHON TO ACCEPT TWO INTEGERS AS AN INPUT. AND FIND WHICH IS THE BIGGEST NUMBER AMONG THE GIVEN TWO NUMBER.