**Expression:**

An expression has a value.

All the following are expressions.

- constant

example : 1729 "rose" 3.14

- variable

var = 1729

var is an expression

- expression binary\_operator expression

3 + 4

- unary operator expression

-5

- expression within parentheses

(3 + 4)

Please note the following.

- An expression has a value

- A statement does not

- An expression is also a statement

- A statement is not necessarily an expression

This is an assignment statement. This is not an expression

a = 10

#print(a = 10) # error

# this is an expression as well as a statement - but not very meaningful.

3 + 4

**operators:**

These indicate some action resulting in a value.

We talk about the following with respect to operators.

1. **arity or rank**

- refers to the number of operands required for the operator

- could be 1 or 2 or 3

2. **precedence**

order of evaluation:

example:

multiplicative operators have a higher precedence compared to additive operators.

2 \* 3 + 4 => 6 + 4 => 10

2 + 3 \* 4 => 2 + 12 => 14

3. **association**

if more than one operator with the same level of precedence,

association indicates the order of evaluation

2 \* 3 \* 4 => 6 \* 4 => 25

\* : multiplicative operator : left associative

2 \*\* 3 \*\* 4 => 2 \*\* 81 => ...

\*\* : exponentian operator : right associative

Let us examine a few operators. Some of these are self explanatory.

**arithmetic operators:**

+ : addition

- : subtraction

\* : multiplication

/ : division

% : remainder ; also called modulo operation

// : integer division

\*\* : exponentiation

Examples:

>>> 25 / 4

6.25

>>> 25 // 4

6

>>> 25 % 4

1

>>> 25 \*\* 3

15625

Find out whats happening in these cases.

print(25 / 4, 25 % 4, 25 // 4, 27 // 4, -25 // 4, -27 // 4)

# 6.25 1 6 6 -7 -7

# % : modulo operator

print(25.8 % 4.2)

# 0.6

**bitwise operator:**

& => and ; result is 1 if the corresponding bits are one

| => or ; result is 1 if even one of the bits is one

^ => exclusive or ; result is 1 if and only if one of the bits is 1

<< => left shift ; multiply by 2 for each left shift

>> => right shift ; divide by 2 for each right shift

~ => one's compliment ; change 0 to 1 and 1 to 0

a = 5 # 0101

b = 6 # 0110

print ("a & b ", a & b) # 0100 => 4

print ("a | b ", a | b) # 0111 => 7

print ("a ^ b ", a ^ b) # 0011 => 3

print ("a << 4 ", a << 4) # 0101 0000 => 80

print ("75 >> 3 ", 75 >> 3) # 0100 1011 >> 3 => 0100 1 => 9

print ( "~ a ", ~a ) # 111111111 .... 1010 => -6

Here is an interesting example of swapping two integers without using extra variable.

Follow the comments.

# file: 1\_bitwise\_swap.py

# interchange two int variables without using another variable

a = 5

b = 6

print("before : ")

print("a : ", a)

print("b : ", b)

a = a ^ b # 0101 ^ 0110 => 0011 => 3

b = a ^ b # 0011 ^ 0110 => 0101 => 5

a = a ^ b # 0011 ^ 0101 => 0110 => 6

print("after : ")

print("a : ", a)

print("b : ", b)

$ python 1\_bitwise\_swap.py

before :

a : 5

b : 6

after :

a : 6

b : 5

**relational operators:**

These are used to compare two values.

The result is of bool type with values True and False.

These are the relation operators.

<<= >>= == != in is

Please check each of the expressions and read the comments carefully.

# file: 2\_relational\_operator.py

# relational operators

# used to compare two quantities

# <<= >>= == != in is

# result : bool

# values : False True

#

# **simple comparison**

print("10 == 10", 10 == 10) # True

print("3 > 2 : ", 3 > 2) # True

# **cascading comparison**

# a op1 b op2 c is same as (a op1 b) and (b op2 c)

# Python knows math better than any other language!!

print("3 > 2 > 1 : ", 3 > 2 > 1)

print("10 == 10 == 10 : ", 10 == 10 == 10)

# a > b > c : (a > b) and (b > c)

# **string comparison:**

# compares the corresponding characters based on the coding - based how the character

# is stored as a number in the computer - until a mismatch or one or both strings end.

print("cat > car : ", "cat">"car") # True # "t">"r"

print("cat > cattle : ", "cat">"cattle" ) # False : second string is longer and therefore bigger

print("cat == Cat : ", "cat" == "Cat") # False : "C"<"c"

print("apple > z : ", "apple">"z") # False ; comparison not based on the length

print("zebra > abcdefgh : ", "zebra">"abcedefgh") # True "z"> a"; rest do not matter

# **list comparison:**

# rule same as that of string - compare the corresponding elements until a mismatch or one or both ends

print([10, 20, 30] > [10, 25]) # False 20 > 25 is false

print([(10, 20), "abcd" ] >[(10, 20), "abcc" ]) # True d of abcd > last c of abcc

# **in : membership**

print("c in cat", "c" in "cat") #True

print("at in cat", "at" in "cat") # True

print("ct in cat", "ct" in "cat") #False

print("ta in cat", "ta" in "cat") #False

print("cat">"cat") # False

print("cat">= "cat") # True

**# logical operators**

# not

# and

# or

a = 10

b = 10

print (not (a == b) ) # False

print(a > 5 and b > 5) # True

print(a > 5 and b < 5) # False

print(a < 5 and b < 5) # False

a = 0

b = 10

#print( b / a > 5) # division by zero

print( a == 0 or b / a > 5)

# **short circuit evaluation**

# evaluate a logical expression left to right

# stop the evaluation as soon as the truth or the falsehood is found

# Observe this is similar to Don't cares in K maps.

**Logical operators:**

These operators not and or operate on boolean values.

**In Python, the following are true.**

True 5 -5 1 "python" ["we", "love", "python"] non-empty-data-structure

**In Python, the following are false.**

False 0 "" [] None empty-data-structure

**# operators and polymorphism:**

Some operators behave differently based on the type of the operands. They exhibit different forms. These are said to be polymorphic.

operator + on numbers is addition operator.

operator + on strings, tuples, lists is concatenation operator - juxtapose the two items.

operator \* on numbers is multiplication operator.

Operator \* on strings, tuples, lists with an integer is replication operator - repeat the elements # of times.

Also, observe that the operator remains commutative even if the operands are switched.

# file: 3\_polymorphic\_operator.py

# polymorphic operator

# +

print(10 + 20) # 30

print("one" + "two") # onetwo # concatenation

print([10, 20] + [30, 40]) # [10, 20, 30, 40] # concatenation

# \*

print(2 \* 3) # 6

print("2" \* 3) # 222 # replicate

print("python" \* 3) # pythonpythonpython

print((10, 20) \* 3) #(10, 20, 10, 20, 10, 20)

print(3 \* "2") # 33 # commutative.