dict Data Type:-

If we want to represent a group of values as key-value paires then we should go for dict datatype.

Eg:- d= {101 : 'Ram', 102: 'Ravi', 103: 'Rohit'}

(*) Duplicate keys are not allowed but values can be duplicated.

(*) If we are trying to insert an entry with duplicate key then old value will be replaced with new value.

Eg!

```
>>> d ={101: 'Ram', 102: 'Ravi', 103: 'Rohit'}
>>> d[101] = 'Rahim'
>>> d
{101 : 'Rahim', 102: 'Ravi', 103 : 'Rohit'}
```

we can create empty dictionary as follows.

```
d={}
```

We can add key-value pair as follows.

```
d['a'] = 'apple'
d['b'] = 'banana'
print(d)
```

(*) dict is mutable and the order wouldn't be preserved.

None Data Type:-

None means Nothing or No value associated

If the value is not available, then to handle such type of cases None introduced. It is something like null value in Java.

Eg:

```
def m1():
a=10
print(m1())
 None
```

Escape Characters:-

In String literals we can use escape characters to associate a special meaning.

```
>>>
>>> S = "Cheese\n Malkist"
>>> print(S)
        Cheese
        Malkist
S = "Cheese\t Malkist"
print(s)
     Cheese   Malkist
S= "This is " Malkist
   Made by cheese,
SyntaxError: invalid Syntax
```

```
>>> s = "This is \" symbol"
>>> print(s)
This is "symbol
```

The following are various important escape characters in Python.

1. \n ==> Newline
2. \t ==> Horizontal tab
3. \r ==> Carriage Return
4. \b ==> Back space
5. \f ==> Form Feed
6. \v ==> Vertical tab
7. \' ==> Single quote
8. \" ==> Double quote
9. \\ ==> back slash symbol

Constants:-

Concept of constant is not applicable in Python.

But it is convention to use only uppercase characters if we don't want to change value.

```
Max_VALUE = 10
```

It is just convention but we can change the value.

Operators:-

Operator is a symbol that performs certain operations.

Python provides the following set of operators:-

1. Arithmatic Operators.
2. Relational Operators or Comparison Operators.
3. Logical operators.
4. Bitwise operators.
5. Assignment operators.
6. Special operators. *

* { 6.1 Membership operators. (in, not in)
6.2 Identity operators. (is, is not)

7. Unary operator (- minus operator)

① Arithmatic Operators:-

(+, -, *, /, %)
↳ Modulo operator

// => Floor Divison

** => Exponent operator or power operator test.py

Eg1

```
a=10
b=2
print('a+b=', a+b)
print('a-b=', a-b)
print('a*b=', a*b)
print('a/b =', a/b )
print('a//b=',' a//b )
print('a%b=', a%b)
print('a**b='). a**b)
```

Execution:

```
python test.py / py test.py

a+b = 12
a-b = 8
a*b = 20
a/b = 5.0
a//b = 5
a%b = 0
a**b = 100
```

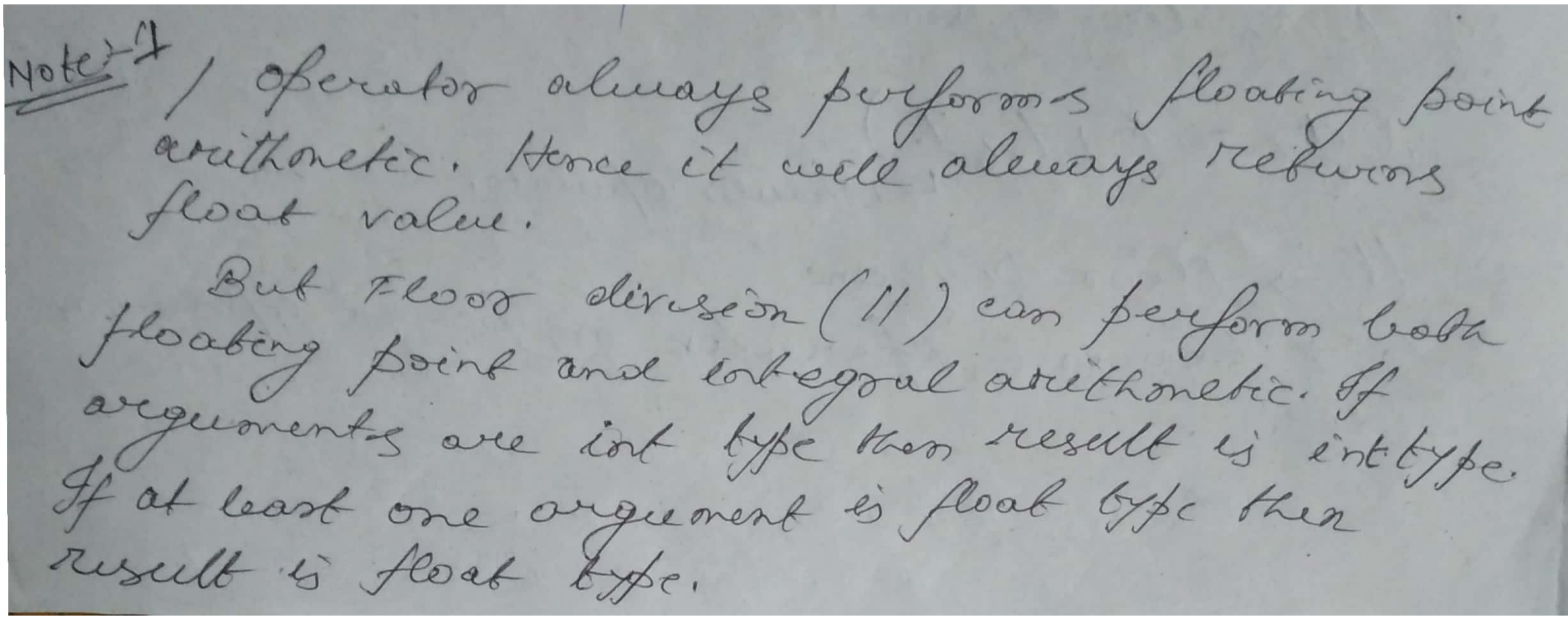
Eg2

```
a = 10.5
b = 2

a+b = 12.5
a-b =8.5
a*b = 21.0
a/b = 5.25
a//b = 5.0
a%b = 0.5
a**b = 110.25
```

Eg3

```
10/2 = 5.0
10//2 = 5
10.0/2 = 5.0
10.0//2 = 5.0
```



Note:-1 / operator always performs floating point arithmetic. Hence it will always returns float value.

But Floor division (//) can perform both floating point and integral arithmetic. If arguments are int type then result is int type. If at least one argument is float type then result is float type.

Note:- 2

We can use +, * operators for str type also.

If we want to use + operator for str type then compulsory both arguments should be str type only otherwise we will get error.

```
>>> [redacted] >>> "Mango" + 10
TypeError : must be str, not int

>>> [redacted] >>> "Mango" + "10"
[redacted]            Mango10
```

Note:- 3

If we use * operator for str type then compulsory one argument should be int and other argument should be str type.

✓ 2 * "Mango"

✓ "Mongo" * 2

2.5 * "Mango"   O/P:- TypeError: can't multiply sequence by non int of type 'float'

"Mango" * "Mango"   O/P:- TypeError: can't multiply sequence by non int of type 'str'

+ => String concatenation operator.

* => String multiplication operator.

Note:- For any number x,

x/0 and x%0 always raises "ZeroDivisionError"

10/0 --- 10.0/0

Relational Operators

(>, >= , <, <=)

Eg:-1

```
a = 10
b = 20
print ("a>b is", a>b)          a>b is False
print ("a>=b is", a>=b)        a>=b is False
print ("a<b", a<b)             a<b is True
print ("a<=b", a<=b)           a<=b is True
```

Eg2:-

```
a= "Mango"
c= "Banana"
b= "Mango"
print ("a>b is", a>b)      o/p:- False
print ("a>=b is", a>=b)    o/p:- True
print ("a<b is", a<b)      o/p:— False
print ("a<=b is" a<=b)     o/p — True
```

Eg3:-

```
print(True > True)    False
print(True >= True)   True
print(10 > True)   True
print(False > True)  False

print(10 > 'Mango')  TypeError: > not supported
                      between instane of 'int' and
                      'str'
```

Eg:4:-

```
   a=10
   b=20
if(a>b):
    print("a is greater than b")
else:
    print("a is not greater than b")
```

output: a is not greater than b

Note:- chaining of relational operators is possible. In the chaining, if all comparisons returns True then only result is True.

If atleast one comparison returns False then the result is False.

Eg:-

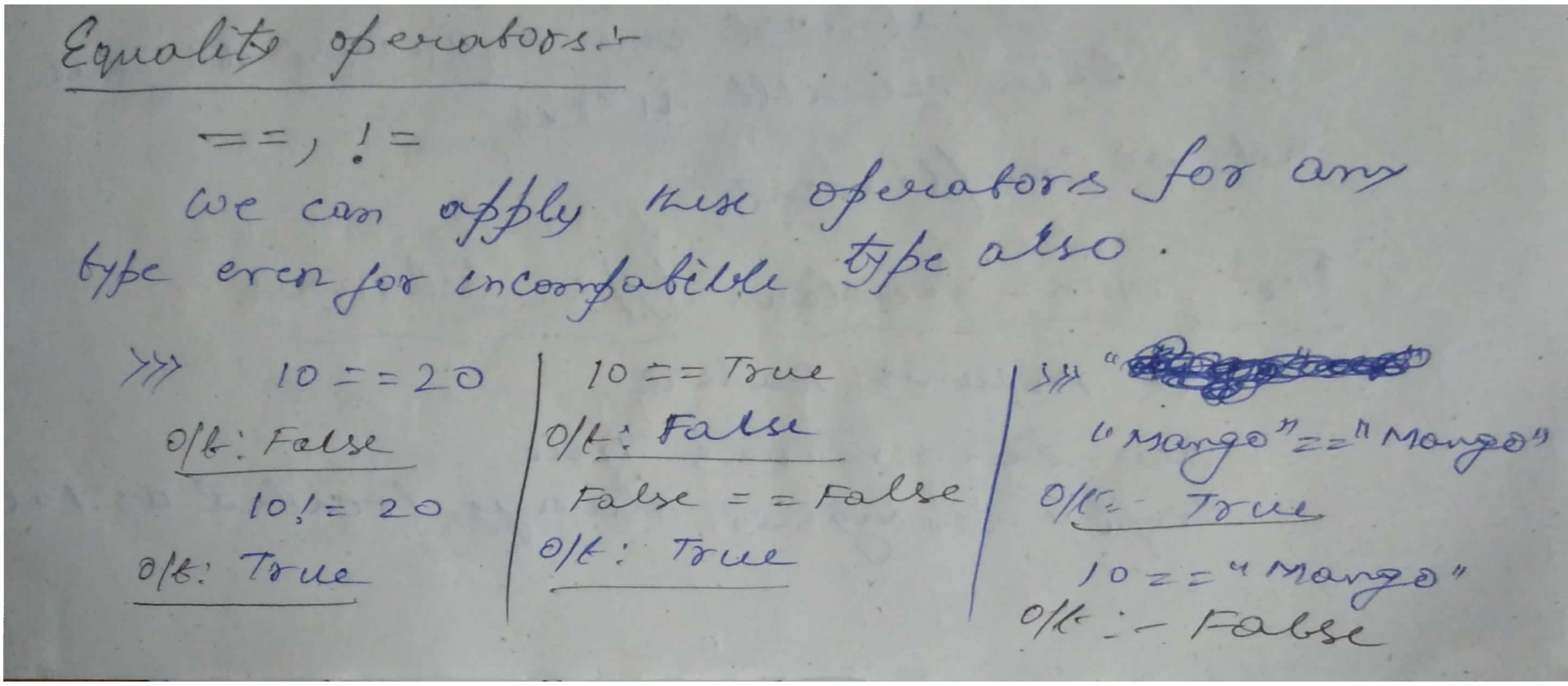
$10 > 20 \Rightarrow$ False

$10 < 20 \Rightarrow$ True

$10 < 20 < 30 \Rightarrow$ True

$10 < 20 < 30 < 40 \Rightarrow$ True

$10 < 20 < 30 < 40 > 50 \Rightarrow$ False



Equality operators:-

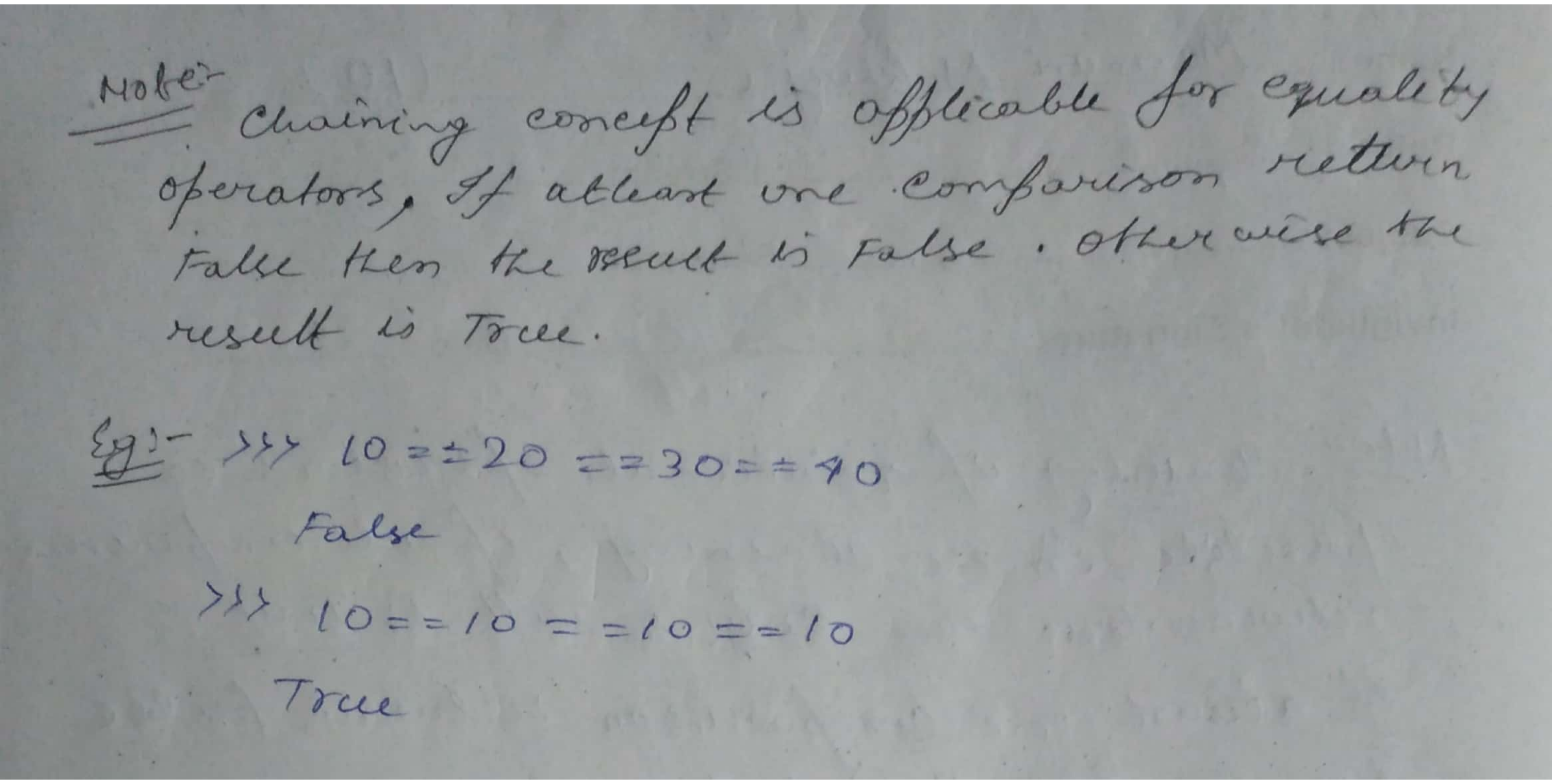
==, !=

We can apply these operators for any type even for incompatible type also.

```
>>> 10 == 20
o/p: False
10 != 20
o/p: True
```

```
10 == True
o/p: False
False == False
o/p: True
```

```
>>> "
"Mango" == "Mango"
o/p:- True
10 == "Mango"
o/p :- False
```



Note:- Chaining concept is applicable for equality operators, If atleast one comparison return False then the result is False. otherwise the result is True.

Eg:-

```
>>> 10==20==30==40
False
>>> 10==10==10==10
True
```

Logical operator:-

and, or, not

We can apply for all type.

For boolean types behaviour:-

and => If both arguments are True then only result is True.

or => If atleast one argument is True then result is True

not => Complement

For non-boolean types behaviour:-

0 means False

non-zero means True

empty string is always treated as False

x and y:

⇒ If x is evalutes to false return x otherwise return y

Eg:- 10 and 20

0 and 20

If first argument is zero then result is zero otherwise result is y.

x or y:

If x evalutes to True then result is x otherwise result is y.

10 or 20 ⇒ 10

0 or 20 ⇒ 20

not x :-

If x is evalutates to False then results is True otherwise False

not 10 => False

not 0 => True

Eg: "Mango" and "Mango juice" => Mangojuice

"" and "Mango" => ""

"Mango" and "" => ""

"" or "Mango" => "Mango"

"Mango or "" => "Mango"

not "" => True

not "Mango" => False

Bitwise Operators:-

⊛ We can apply these operators bitwise.

⊛ These operators are applicable only for int and boolean types.

⊛ By mistake if we are trying to apply for any other type then we will get error.

(right shift)

(&, |, ^, ~, <<, >>)

and (or) (ex-or) (not) (Leftshift)

print(4&5) ==> (100 & 101) => 100 => 4 (valid)

print(10.5 &5.6) ==> TypeError: unsupported operand type(s), for & : 'float' and 'float'

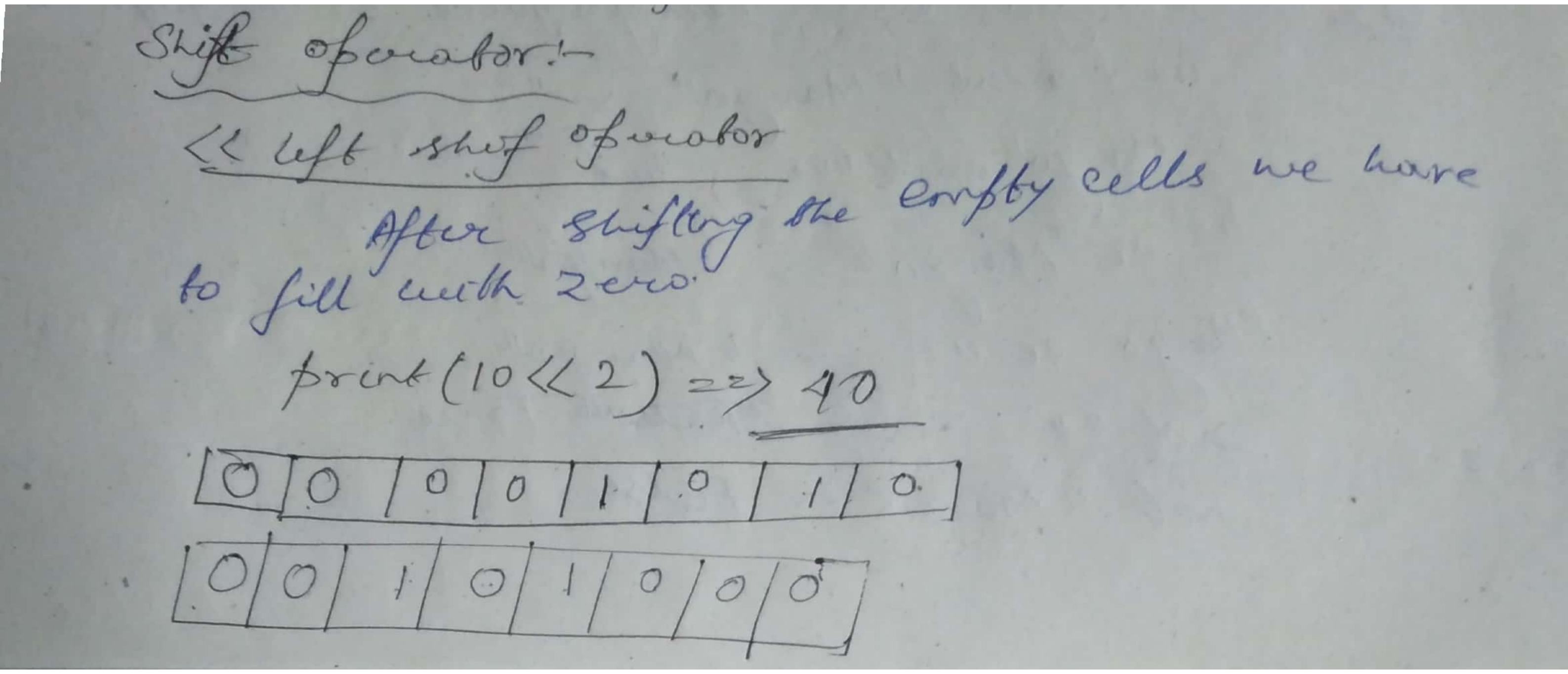
print (True & True) ==> True (valid)

Eg) print (4|5) ==> (100|101) => 101 => 5

print(4^5) ==> (100^101) => (001) => 1

Bitwise (ex-or) (If bits are different then only result is 1) otherwise is 0

0101 .



Shift operator:-

<< left shift operator

After shifting the empty cells we have to fill with zero.

print(10<<2) => 40

0	0	0	0	1	0	1	0

0	0	1	0	1	0	0	0

>> Right Shift Operator:-

After shifting the empty cells we have to fill with sign bit.(0 for +ve and 1 for -ve)

print(10>>2) ==2

0	0	0	0	1	0	~~1~~	~~0~~

↓

0	0	0	0	0	0	1	0

We can apply bitwise operators for boolean types also:-

print(True &False) => False

print(True|False) => True

print(True ^ False) => True

print(~True) => -2

print(True <<2) => 4

print(True >>2) => 0