

3.1 Operators

- Logical
- Equality
- Comparison
- Arithmetic
- Bitwise

3.1.1 Logical Operators

In python following keywords are used for boolean operations -

Keywords	Meaning
not	unary negation
and	conditional AND
or	conditional OR

EXAMPLES:

In [1]:

```
True and True
```

Out[1]:

True

In [2]:

```
True or False
```

Out[2]:

True

In [4]:

```
# Lets define two variables having boolean values True and False-
START = True
STOP = False

# Print the values as it is
print(f"Defined value of START = {START}")
print(f"Defined value of STOP = {STOP} \n")

# if can also be written as -
print(f"Defined value of START = {START}")
print(f"Value of STOP by negating START= {not START} \n") #because not of True is Flase

# alternatively -
print(f"Value of START by negating STOP = {not STOP}")
print(f"Defined value of STOP = {STOP} \n")
```

Defined value of START = True
Defined value of STOP = False

Defined value of START = True
Value of STOP by negating START= False

Value of START by negating STOP = True
Defined value of STOP = False

In [5]:

```
START = 1
STOP = 0

# Print the values as it is
print(f"Defined value of START = {START}")
print(f"Defined value of STOP = {STOP} \n")

# if can also be written as -
print(f"Defined value of START = {START}")
print(f"Value of STOP by negating START= {int(not START)} \n")

# alternatively -
print(f"Value of START by negating STOP = {int(not STOP)}")
print(f"Defined value of STOP = {STOP} \n")
```

Defined value of START = 1
Defined value of STOP = 0

Defined value of START = 1
Value of STOP by negating START= 0

Value of START by negating STOP = 1
Defined value of STOP = 0

In [7]:

```

#imp example
zero = 0
one = 1

print(f"boolean value of no. {zero} is {bool(zero)}")
print(f"boolean value of no. {one} is {bool(one)}")
print(f"negation of {zero} is {not zero} and negation of {one} is {not one}")
print("\n#-----#\n")

Some_negative_integer = -5
Some_positive_integer = 5

print(f"boolean value of no. {Some_negative_integer} is {bool(Some_negative_integer)}")
print(f"boolean value of no. {Some_positive_integer} is {bool(Some_positive_integer)}")
print(f"negation of {Some_negative_integer} is {not Some_negative_integer} \
and negation of {Some_positive_integer} is {not Some_positive_integer}")

print("\n#-----#\n")

Some_negative_float = -5.99
Some_positive_float = 5.6

print(f"boolean value of no. {Some_negative_float} is {bool(Some_negative_float)}")
print(f"boolean value of no. {Some_positive_float} is {bool(Some_positive_float)}")
print(f"negation of {Some_negative_float} is {not Some_negative_float} \
and negation of {Some_positive_float} is {not Some_positive_float}")

"""
Conclusion
bool(0)=False
bool(1)=True
bool(any other no.)=True
int(bool(any other no.))=1
"""

```

```

boolean value of no. 0 is False
boolean value of no. 1 is True
negation of 0 is True and negation of 1 is False

```

```
#-----#
```

```

boolean value of no. -5 is True
boolean value of no. 5 is True
negation of -5 is False and negation of 5 is False

```

```
#-----#
```

```

boolean value of no. -5.99 is True
boolean value of no. 5.6 is True
negation of -5.99 is False and negation of 5.6 is False

```

Out[7]:

```

'\nConclusion\nbool(0)=False\nbool(1)=True\nbool(any other no.)=True\nint
(bool(any other no.))=1\n'

```

In [9]:

```
#Observation  
False*False
```

Out[9]:

0

In [10]:

```
not False *False
```

Out[10]:

True

In [11]:

```
not(False)*False
```

Out[11]:

True

In [17]:

```
print((not(False))*False)
```

0

In [14]:

```
True * False
```

Out[14]:

0

#not is a boolian operation therefore always gives answer in True and False

Above example in tabular format-

VEGETABLES	SALT	DISH
False	False	False
False	True	False
True	False	False
True	True	True

Above table represents AND gate's Truth table-

X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

In [23]:

```
VEGETABLES = True
SALT = False
DISH = VEGETABLES and SALT

print(f"Dish contains VEGETABLES: {VEGETABLES}")
print(f"Dish contains SALT: {SALT}")
print(f"Hence dish prepared was good: {DISH}\n")

VEGETABLES = True
SALT = True
DISH = VEGETABLES and SALT

print(f"Dish contains VEGETABLES: {VEGETABLES}")
print(f"Dish contains SALT: {SALT}")
print(f"Hence dish prepared was good: {DISH}\n")
```

Dish contains VEGETABLES: True
Dish contains SALT: False
Hence dish prepared was good: False

Dish contains VEGETABLES: True
Dish contains SALT: True
Hence dish prepared was good: True

Above example in tabular format-

BIKE	CAR	TRAVEL_100_KM
False	False	False
False	True	True
True	False	True
True	True	True

Above table represents OR gates Truth table-

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

In [24]:

```
BIKE = True
CAR = True
TRAVEL_100_KM = BIKE or CAR

print(f"You have BIKE: {BIKE}")
print(f"You have CAR: {CAR}")
print(f"You can travel 100 KMs: {TRAVEL_100_KM}")
```

You have BIKE: True
 You have CAR: True
 You can travel 100 KMs: True

In [26]:

```
BIKE = False
CAR = True
TRAVEL_100_KM = BIKE or CAR

print(f"You have BIKE: {BIKE}")
print(f"You have CAR: {CAR}")
print(f"You can travel 100 KMs: {TRAVEL_100_KM}")
```

You have BIKE: False
 You have CAR: True
 You can travel 100 KMs: True

3.1.2 Equality Operators

Following operations are present in python for equality check operation-

Operators	Meaning
is	<i>a is b</i> returns true if variable/identifiers a and b <i>points to the same object</i>
is not	<i>a is not b</i> returns true if variable/identifiers a and b <i>points to the different object</i>
==	<i>a == b</i> returns true if variable/identifiers a and b has same value
!=	<i>a != b</i> returns true if variable/identifiers a and b has different value

In [28]:

```
list_a = [1,2,3]
list_b = [1,2,3]

print(f"id of list_a = {id(list_a)}")
print(f"id of list_b = {id(list_b)}")
print(f"list_a is list_b = {list_a is list_b}")
print(f"list_a == list_b = {list_a == list_b}")
```

```
id of list_a = 2581863767296
id of list_b = 2581829019072
list_a is list_b = False
list_a == list_b = True
```

In above example we can observe that even though the values in the list are same they are stored in different addresses that is why list_a is list_b = False

In [32]:

```
# to solve the above problem Let
list_c = list_a

print(f"id of list_a = {id(list_a)}")
print(f"id of list_c = {id(list_c)}")
print(f"list_a is list_c = {list_a is list_c}")
print(f"list_a == list_c = {list_a == list_c}")
#both lists will have same address i.e list_a is list_c = True
```

```
id of list_a = 2581863767296
id of list_c = 2581863767296
list_a is list_c = True
list_a == list_c = True
```

NOTE

this concept does not apply to string cuz string are immutable while lists are mutable

In [36]:

```
#example
a = 6
b = 6
print(id(a))
print(id(b))
```

```
2581745330640
2581745330640
```

In [37]:

```
str1 = "asit"
str2="asit"
print(id(str1))
print(id(str2))
```

```
2581873702448
2581873702448
```

In [38]:

```
str1 = "asit"
str2="asit1"
print(id(str1))
print(id(str2))
#different attresses
```

```
2581873702448
2581864544240
```

In [39]:

```
#immutability of strings
str1[1]
```

Out[39]:

```
's'
```

In [42]:

```
str1[1]='b'

"""only traversing through is allowed and changing the character
of the string is not allowed therefor string is immutable"""
```

```
-----
-
TypeError                                Traceback (most recent call las
t)
~\AppData\Local\Temp\ipykernel_6488\3614166174.py in <module>
----> 1 str1[1]='b'
      2 """only traversing through is allowed and changing the character
      3 of the string is not allowed therefor string is immutable"""
```

TypeError: 'str' object does not support item assignment

In [45]:

```
#Mutability of Lists
print(list_a)
list_a[1]=3
print(list_a)
```

```
[1, 2, 3]
[1, 3, 3]
```


3.1.3 Comparison Operators

Operation	Meaning
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

In [47]:

```
temp_today = 25
temp_yesterday = 27
predicted_temp_nextDay = 27

print(f"today's temperature is less than or equal to yesterday's: {temp_today <= temp_yesterday}")

today's temperature is less than or equal to yesterday's: True
```

3.1.4 Arithmetic Operators

Operation	Meaning
+	addition
-	subtraction
*	multiplication
/	true division
//	integer division
%	the modulo operator

3.1.5 Bitwise Operators

Operation	Meaning
~	bitwise complement (prefix unary operator)
&	bitwise and
	bitwise or
^	bitwise exclusive-or
<<	shift bits left, filling in with zeros
>>	shift bits right, filling in with sign bit

x	y	&
0	0	0
0	1	0
1	0	0

x	y	&
1	1	1

x	y	
0	0	0
0	1	1
1	0	1
1	1	1

x	y	^
0	0	0
0	1	1
1	0	1

In [52]:

```
var_a = 10
binary_a = bin(var_a)

complement_a = ~var_a
binary_complement_a = bin(complement_a)

print(f"var_a: {var_a} binary of var_a: {binary_a}")
print(f"complement of var_a: {complement_a} \
binary of complement of var_a: {binary_complement_a}") # returns one's compliment
```

```
var_a: 10 binary of var_a: 0b1010
complement of var_a: -11 binary of complement of var_a: -0b1011
```

In [53]:

```
var_a = 10
var_b = 4

binary_a = bin(var_a)
binary_b = bin(var_b)

bitwise_and = var_a & var_b
binary_bitwise_and = bin(bitwise_and)

print(f"var_a: {var_a}, var_b: {var_b}")
print(f"binary of var_a: {binary_a}, binary of var_b: {binary_b}")

print(f"bitwise_and: {bitwise_and}, binary_of_bitwise_and: {binary_bitwise_and}")
```

```
var_a: 10, var_b: 4
binary of var_a: 0b1010, binary of var_b: 0b100
bitwise_and: 0, binary_of_bitwise_and: 0b0
```

In [54]:

```
var_a = 10
var_b = 4

binary_a = bin(var_a)
binary_b = bin(var_b)

bitwise_or = var_a | var_b
binary_bitwise_or = bin(bitwise_or)

print(f"var_a: {var_a}, var_b: {var_b}")
print(f"binary of var_a: {binary_a}, binary of var_b: {binary_b}")

print(f"bitwise_or: {bitwise_or}, binary_bitwise_or: {binary_bitwise_or}")
```

var_a: 10, var_b: 4
binary of var_a: 0b1010, binary of var_b: 0b100
bitwise_or: 14, binary_bitwise_or: 0b0

In [55]:

```
var_a = 10
var_b = 4

binary_a = bin(var_a)
binary_b = bin(var_b)

bitwise_xor = var_a ^ var_b
binary_bitwise_xor = bin(bitwise_xor)

print(f"var_a: {var_a}, var_b: {var_b}")
print(f"binary of var_a: {binary_a}, binary of var_b: {binary_b}")

print(f"bitwise_xor: {bitwise_xor}, binary_bitwise_xor: {binary_bitwise_xor}")
```

var_a: 10, var_b: 4
binary of var_a: 0b1010, binary of var_b: 0b100
bitwise_xor: 14, binary_bitwise_xor: 0b0

In [56]:

```
var_a = 50

binary_a = bin(var_a)

right_shift = var_a >> 1
binary_right_shift = bin(right_shift)

print(f"var_a: {var_a}")
print(f"binary of var_a: {binary_a}")

print(f"right_shift: {right_shift}")
print(f"binary_right_shift: {binary_right_shift}")
```

var_a: 50
binary of var_a: 0b110010
right_shift: 25
binary_right_shift: 0b11001

In [58]:

```
# compare above results with interger division
N = 1
var_a // 2**N == var_a >> N
```

Out[58]:

True

In [59]:

```
N = 2
var_a // 2**N == var_a >> N
```

Out[59]:

True

In [60]:

```
var_a = 5
N = 2
var_a // 2**N == var_a >> N
```

Out[60]:

True

In [61]:

```
var_a = 48

binary_a = bin(var_a)

left_shift = var_a << 1
binary_left_shift = bin(left_shift)

print(f"var_a: {var_a}")
print(f"binary of var_a: {binary_a}")

print(f"left_shift: {left_shift}")
print(f"binary_left_shift: {binary_left_shift}")
```

```
var_a: 48
binary of var_a: 0b110000
left_shift: 96
binary_left_shift: 0b1100000
```

In [62]:

```
# compare the above results with multiplication by powers of 2
var_a = 5
N = 2
var_a * 2**N == var_a << N
```

Out[62]:

True

In [63]:

```
# compare the above results with multiplication by powers of 2
var_a = 25
N = 2
var_a * 2**N == var_a << N
```

Out[63]:

True

In []:

3.2 Operators for Sets and Dictionaries

Operation	Meaning
key in s	containment check
key not in s	non-containment check
s1 == s2	s1 is equivalent to s2
s1 != s2	s1 is not equivalent to s2
s1 <= s2	s1 is subset of s2
s1 < s2	s1 is proper subset of s2
s1 >= s2	s1 is superset of s2
s1 > s2	s1 is proper superset of s2
s1 s2	the union of s1 and s2
s1 & s2	the intersection of s1 and s2
s1 - s2	the set of elements in s1 but not s2
s1 ^ s2	the set of elements in precisely one of s1 or s2

Operations on Dictionary

In []:

```
"x" in {"x":25, "y":34}
```

Out[38]:

True

In []:

```
"z" in {"x":25, "y":34}
```

Out[39]:

False

In []:

```
"z" not in {"x":25, "y":34}
```

Out[40]:

True

In []:

Operations on Sets

In []:

```
set_1 = {1,2,3}
set_2 = {1,2,5}

set_1 == set_2
```

Out[41]:

False

In []:

```
set_1 == set_1
```

Out[42]:

True

In []:

```
set_1 != set_2
```

Out[43]:

True

```
set_1 ⊆ set_2
```

In []:

```
set_1 ⊆ set_2
```

Out[44]:

False

In []:

```
set_1 = {1,2,3}  
set_2 = {1,2}  
  
set_2 <= set_1
```

Out[45]:

True

$\text{set_1} \subset \text{set_2}$

In []:

```
set_2 < set_1
```

Out[46]:

True

$\text{set_1} \supseteq \text{set_2}$

In []:

```
set_2 >= set_1
```

Out[48]:

False

In []:

```
set_1 >= set_2
```

Out[49]:

True

$\text{set_1} \supset \text{set_2}$

In []:

```
set_1 > set_2
```

Out[50]:

True

```
set_1 ∪ set_2
```

In []:

```
set_1 | set_2
```

Out[51]:

```
{1, 2, 3}
```

```
set_1 ∩ set_2
```

In []:

```
set_1 & set_2
```

Out[52]:

```
{1, 2}
```

```
set_1 - set_2
```

In []:

```
set_1 - set_2
```

Out[53]:

```
{3}
```

```
set_1 ^ set_2
```

In []:

```
set_1 ^ set_2
```

Out[54]:

```
{3}
```


3.3 Extended Assignment Operators

```
In [ ]:
alpha = [1, 2, 3]
beta = alpha # an alias for alpha
beta += [4, 5] # extends the original list with two more elements
beta = beta + [6, 7] # reassigns beta to a new
print(alpha)
```

3.4 Operator Precedence

Following table represents operators from highest to the lowest precedence -

Operator Precedence		
Precedence	Type	Symbols
1	member access	expr.member
2	function/method calls	expr(...)
3	container subscripts/slices	expr[...]
4	exponentiation	**
5	unary operators	+expr, -expr, ~expr
6	multiplication, division	*, /, //, %
7	addition, subtraction	+, -
8	bitwise shifting	<<, >>
9	bitwise-and	&
10	bitwise-xor	^
11	bitwise-or	
12	comparisons	is, is not, ==, !=, <, <=, >, >=
13	containment	in, not in
14	logical-not	not expr
15	logical-and	and
16	logical-or	or
17	conditional	val1 if condition else val2
18	assignments	=, +=, -=, etc.

Reference:-

```
@book{goodrich2013data,  
      title={Data Structures and Algorithms in Python},
```