Session 3

- len() function
- Indexing continued
- Slicing
- Operators in Python
  - Arithmetic Operator
  - Assignment Operator
  - Logical Operator
  - Bitwise Operator
  - Operator Precedence

1) len() function

`len()` function is a built0in python method which returns us the lenght/count of the number of elements in a collection datatype(lists,strings,dictionary etc.)
The `len()` when called on nested multi dimensional list only the number of elements in the first dimension is returned.

In [5]:
```
my_list = [1,2,3]
print(my_list)
print(len(my_list))
```

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

In [6]:
```
my_list = ['A string',23,100.232,'o']
print(my_list)
print(len(my_list))
```

```
['A string', 23, 100.232, 'o']
4
```

In [7]:
```
my_list = [[1,2,3],[4,5,6],[7,8,9]]
print(my_list)
print(len(my_list))
```

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
3
```

NOTE : While using the len function even the spaces in strings are counted as an element

In [9]:
```
x = "hello world"
print(len(x))
```

```
11
```

TASK 1

```
my_list = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32,
34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54,
           56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84,
86, 88, 90, 92, 94, 96, 98]
```

- **Try accessing the last element of the above list without counting the numbrer of elements**
  Hint : use len() function

2) Indexing continued

we can use negative indexing to get the elements from the last of a list or collection datatype

```
list elements : 11  22  33  44  55  66
index numbers : -6  -5  -4  -3  -2  -1
```

In [19]:
```
my_list = [11,22,33,44,55,66]
my_list[-2]
```

Out[19]:
```
55
```

3) Slicing

syntax

`my_list[start index : stop index : step size]`

step size is 1 by default

In [21]:
```
my_list = [11,22,33,44,55,66,77,88,99]
my_list[2:5:1]
```

Out[21]:
```
[33, 44, 55]
```

In [23]:
```
my_list = [11,22,33,44,55,66,77,88,99]
my_list[2:7]
```

Out[23]:
```
[33, 44, 55, 66, 77]
```

Note : the last element is always the `(index-1)`

In [24]:
```
my_list = [11,22,33,44,55,66,77,88,99]
my_list[2:7:2]
```

Out[24]: [33, 55, 77]

We can skip the first, last element or both.

In [26]:
```
my_list = [11,22,33,44,55,66,77,88,99]
my_list[:7:1]                          ##skipping first element means we want all the
                                       ##from the start to the specified last index n
```

Out[26]: [11, 22, 33, 44, 55, 66, 77]

In [27]:
```
x = "hello world"
print(x[5:])
```

```
 world
```

TASK 2

Try using negative index inisde slicing syntax

4) Operators in Python

1. Arithmetic Operator
2. Assignment Operator
3. Logical Operator
4. Bitwise Operator
5. Operator Precedence

Arithmatic Operators

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Addition, Multiplication, subtraction and division are the standard operations.

The Modulus (%) operator gives us the remainder of a division operation

In [1]:
```
a = 10
```

```
b = 2

print(a%b)
```

```
0
```

In [2]:
```
a = 10
b = 3

print(a%b)
```

```
1
```

The Exponential (`**`) operator gives us the x raised to y output of 2 numbers (x^y)

In [3]:
```
a = 8
b = 2
print(a**b)
```

```
64
```

We can also calculate squareroots using the exponential operator

In [35]:
```
16**(1/2)
```

Out[35]:
```
4.0
```

In [4]:
```
4**2
```

Out[4]:
```
16
```

The floor division (`//`) operator is used to return the quotient of a division operator without the decimal places

In [5]:
```
a = 8
b = 2
print(a//b)
```

```
4
```

In [6]:
```
a = 8
b = 3
print(a//b)
```

```
2
```

Assignment Operators

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3

The `=` sign is used to assign a datatype to a variable

In [12]:
```
a = 10
print(a)
```

```
10
```

In [13]:
```
a += 10
print(a)
```

```
20
```

The above line of code is similar to `a = a + 10`

In [14]:
```
a = a + 10
print(a)
```

```
30
```

The same logic apply to all the assignment operators

Logical Operators

- and
- or
- not

Gives the oputput depending on the truth table or logic table

A	B	A AND B	A OR B	NOT A
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

We can try to emulate the truth table above by doing the same operations

In [4]:
```
True and True
```

Out[4]: True

In [16]:
```
False and True
```

Out[16]: False

In [17]:
```
False or True
```

Out[17]: True

In [18]:
```
not True
```

Out[18]: False

But the logical operators are never usde to directly operate on boolean values.
Instead we generate these boolean values from comparison operators most of the time

In [18]:
```
not b>=a
```

Out[18]: False

In [15]:
```
not True
```

Out[15]: False

Comparison Operator

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Comparison operator generate Boolean output depending on wheather condition being checked

In [20]:
```
a = 10
b = 20
c = 10

print(a==b)
print(a==c)
print(a!=b)
```

```
False
True
True
```

In [24]:
```
a = 10
b = 10

print(a<b)
print(a>b)
print(a<=b)
print(a>=b)
```

```
False
False
True
True
```

Bitwise operators

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Bitwise operators are also like logical operators they give output based on the truth table. But instead of acting direclty on the Boolean datatype they return results by operating on the Bit representation of the integer,float or string datatype.

- True is considered as the bit `1`
- False is considered as the bit `0`

In [19]:
```
a = 5   #0101       - Bitwise representation of integer 5
b = 10  #1010       - Bitwise representation of integer 10
a&b     #0000       - Bitwise and(&) of bit representation of 5 and 10 flowing the trut
```

Out[19]:
```
0
```

In [1]:
```
a = 5   #0101       - Bitwise representation of integer 5
b = 10  #1010       - Bitwise representation of integer 10
a|b     #1111       - Bitwise or(|) of bit representation of 5 and 10 flowing the truth
```

Out[1]: 15

In [4]:
```
a =  5   #0101       - Bitwise representation of integer 5
a = ~5
```

We might expext the output of `a = ~5` to be `10`

```
a =  5   #0101
a = ~5   #1010      - Bitwise not(~) of bit representation of 5
```

Which is the bit representation of 10. But this is **not true**

When we do Bitwise not(~) in python 2 things are happening

- The positive number gets converted to negative number
- The 2's complement of the number is returned

In [5]:
```
a =  5   #0101       - Bitwise representation of integer 5
a = ~5
a
```

Out[5]: -6

It is totally ok if You don't undersatnd the Bitwise not operator. Only thing that you must remember is that doing a bitwise operation does not simply return the Decimal representation of the inverted Bitwise number.

The left shift and right shift operator shifts the bits in the bit representation by number of positions specified

The Right shift operation on bit representaion of 5 by 1 place

In [2]:
```
a = 5   #0101
a>>1    #0010
```

Out[2]: 2

The Left shift operation on bit representaion of 5 by 1 place

In [4]:
```
a = 5   #0101
a<<1    #1010
```

Out[4]: 10

Home Work

1) create a basic calculator

- Take 2 numbers from the user (using input statement)
- Print out the remainder of 2 numbers
- Print out the quotient with decimal places.
- and the bitwise and of the numbers

In [ ]:

2) Guess the output of the below operation

1. `3<10 and 4<5 or False`

In [ ]:

1. `not (15>10 and 17<8)`

In [ ]:

1. `3<10 or 4<9 or 5>=9 and False`

In [ ]: