Session 4

Day 3: Python Programming

- Mutability:

Let us understand with the help of following example:
Consider a list:

```
a=[10,20,30,40]
print(id(a))
print(a)
a[0]=100
print(id(a))
```

Output:

```
4338796864
[10, 20, 30, 40]
4338796864
```

Here, id of 'a' is same in both cases which is not possible in immutable. Thus, in the case of mutable, no new object is created as in immutable.

Since we have already covered fundamental data types: int, float, complex, bool and str. Each of these fundamental data types hold a single value: 'a', 10, 10.5, True, 5+1j, etc.
Now, moving forward to the collection-related types of dataset which hold a group of values.

The collection type dataset are:

- list
- tuple
- set
- frozenset
- dict
- range
- byte
- bytesarray

1. list:

If we want to represent a group of values as a single entity where insertion order is preserved and duplicates are allowed, we should go for a list.
List is mutable in Python.

Example:

```
a=[10, 20, 30, 40, 10]
print (a)
print(type(a))
```

Output:

```
[10, 20, 30, 40, 10]
<class 'list'>
```

Heterogenous objects are allowed along with indexing and slicing concepts in list. For example:

```
a=[10,20,30,True, "Hello", 2+5j]
print(a)
print(type(a))
print(a[1])
print(a[2:4])
a[4]= "Hi"
a.append(20)    #adds 20 to the end of the list preserving
the order of insertion
print(a)
a.remove(20)  #removes the first occurrence
print(a)

#if we want to print each element in the list one at a time,
we use looping structure
for x in a:
    print(x)
```

Output:

```
[10, 20, 30, True, 'Hello', (2+5j)]
<class 'list'>
20
[30, True]
[10, 20, 30, True, 'Hi', (2+5j), 20]
[10, 30, True, 'Hi', (2+5j), 20]
10
30
True
Hi
(2+5j)
20
```

2. tuple:

They are exactly as list but they are immutable. Small brackets() are used in tuples.

For example:

```
t= (10, 20, 30, 40)
print(t)
print(type(t))
```

Output:

```
[10, 20, 30, 40, 10]
<class 'tuple'>
```

But, if:

```
t= (10)
print(type(t))
```

Output:

```
<class 'int'>
```

If an arithmetic expression needs priority, we use parentheses. Python also understands this to be an arithmetic expression. Thus, to add one value in tuple, we need to add a ‘,’ after the value inside the parentheses.

For example:

```
t=(10,)
print(type(t))
```

Output:

```
<class 'tuple'>
```

Here, indexing and slicing concepts are applicable as well.

For example:

```
t=(10,20,30,40)
print(t)
print(t[1])
print(t[1:4])
t[2]= 200
print(type(t))
```

Output:

```
(10, 20, 30, 40)
20
(20, 30, 40)
```

```
    t[2]= 200
    ^^^^
TypeError: 'tuple' object does not support item
assignment
```

Here, item assignment is not supported by tuple objects, hence the error.

Note: For applicability of list and tuple: choose list, if changes are a part of the system and choose tuple, if there needs to be no changes.

3. set:

If we want to represent a group of values without duplicates where insertion order is not preserved, we should go for set.
For example:

```
s= {10,20,30}
print(type(s))
s={1}
print(type(s))
s={}
print(type(s))
#if a single value set is to be created, we use the
function:
s=set()
print(type(s))
```

Output:

```
<class 'set'>
<class 'set'>
<class 'dict'>
<class 'set'>
```

Set is mutable in python.
Here , heterogeneous objects are allowed but indexing and slicing concepts are not.

For example:

```
s={10,20.5,30}
s.add(40)    #since insertion order is not preserved, we
don't need to use append
print(s)
s.remove(30)
print(s)
```

Output:

```
{40, 10, 20.5, 30}
{40, 10, 20.5}
```

Since, elements in a set can be removed and added, it is mutable.

4. **frozenset**:

This is exactly as set but it is immutable in nature, so no adding or removing is possible without new object creation. It is basically freezing a set.

For example:

```
s={10}
print(type(s))
fs=frozenset(s)
print(type(fs))
fs.add(20)
for i in fs:
    print(i)
```

Output:

```
<class 'set'>
<class 'frozenset'>

   fs.add(20)
   ^^^^^^
AttributeError: 'frozenset' object has no attribute
'add'
```

Note: ‘tuple’ and ‘frozenset’ are both immutable, a basic difference on how to choose between them are as follows:

tuples	**frozenset**
Order is important.	Order is not important.
Duplicates are allowed.	Duplicates are not allowed.
Indexing and slicing concepts are applicable.	Indexing and slicing concepts are not applicable.

5. dict:

If a group of values need to be represented in a key value pair, we should go for dict (dictionary) type.
For example:

```
d= {100: 'Thapa' , 200: "Vaskar"}
print(type(d))
print(d)
d[300]= 'Isha'
print(d)
d[200]= 'isha'
print(d)
```

Output:

```
<class 'dict'>
{100: 'Thapa', 200: 'Vaskar'}
{100: 'Thapa', 200: 'Vaskar', 300: 'Isha'}
{100: 'Thapa', 200: 'isha', 300: 'Isha'}
```

Here, we can see that dict is mutable in nature. Here, insertion order is not preserved and slicing and indexing concept is also not applicable. Duplicates are allowed (but if a key has a duplicate value, then it simply replaces the value with the latest value).

6. range:

This data type represents a sequence of numbers. It is immutable in nature.
It can be written in different forms:

Form 1:
range(10) – 0 to 9

Form 2:
range(10, 20) – 10 to 19

Form 3:
range(10, 20, 2) – 10, 12, 14, 16, 18

For example:

```
r= range(10)
print(r)
print(type(r))
for i in r:
```

```
    print(i)
r= range(10,20)
print(r)
print(type(r))
for i in r:
    print(i)
r= range(10,20,2)
print(r)
print(type(r))
for i in r:
    print(i)
r= range(20,10,-3)
print(r)
print(type(r))
for i in r:
    print(i)
```

Output:

```
range(0, 10)
<class 'range'>
0
1
2
3
4
5
6
7
8
9
range(10, 20)
<class 'range'>
10
11
12
13
14
15
16
17
18
```

```
19
range(10, 20, 2)
<class 'range'>
10
12
14
16
18
range(20, 10, -3)
<class 'range'>
20
17
14
11
```

Here, indexing and slicing is applicable. Value modification and item assignment is not supported.

For example:

```
r=range(20,10,-1)
print(r[0])
print(r[2:5])
print(r[0]=1)
```

Output:

```
20
range(18, 15, -1)
```

But,

```
r=range(20,10,-1)
print(r[0]=1)
```

Output:

```
   print(r[0]=1)
         ^^^^^
SyntaxError: expression cannot contain assignment,
perhaps you meant "=="?
```

7. bytes:

It is the collection of bytes. If binary files need to be handled, bytes are used. It is similar to arrays.

For example:

```
b=[10,20,30,40]
by=bytes(b)
print(type(by))
```

Output:

```
<class 'bytes'>
```

Bytes are immutable in nature.
Here, indexing and slicing is possible. But, heterogeneous objects are not possible. Also, the range supported by bytes is 0 to 255, so any value above 255 is not supported.

For example:

```
b=[10,20,30,40]
by=bytes(b)
print(type(by))
print(by[0])            #indexing possible
print(by[0:3])          #slicing possible, shows object
b=[10,20,30,40,256]
by=bytes(b)
for i in by:
    print(i)
```

Output:

```
<class 'bytes'>
10
b'\n\x14\x1e'

by=bytes(b)
^^^^^^^^
ValueError: bytes must be in range(0, 256)
```

8. bytesarray:

It is exactly the same as bytes but it is mutable in nature.

For example:

```
b=[10,20,30,40]
ba=bytearray(b)
print(type(ba))
for i in ba:
    print(i)
ba[0]=100
for i in ba:
    print(i)
```

Output:

```
<class 'bytearray'>
10
20
30
40
100
20
30
40
```

Both byte and bytesarray are internally used by python to handle images. We don't normally use these data types, but it can be used to encrypt passwords, generation of hash codes, etc.

9. None:

None data type is used if an object needs to be made eligible for garbage collection. In case of functions, with no return, then internally that function returns 'None'.

For example:

```
def f():
  a=10
```

In the above function, 'None' is returned internally.