Firstly, the syntax for bytes literals is largely the same as that for string literals, except that a b prefix is added:

- Single quotes: b'still allows embedded "double" quotes'
- Double quotes: b"still allows embedded 'single' quotes"
- Triple quoted: b'''3 single quotes''', b"""3 double quotes"""

Only ASCII characters are permitted in bytes literals (regardless of the declared source code encoding). Any binary values over 127 must be entered into bytes literals using the appropriate escape sequence.

**Example:**

```
>>> b1=b'ABC'
>>> print(b1,type(b1))
b'ABC' <class 'bytes'>
>>> print(b1[0],b1[1],b1[2])
65 66 67
>>> b2=b"abc"
>>> print(b2,type(b2))
b'abc' <class 'bytes'>
>>> print(b2[0],b2[1],b2[2])
97 98 99
>>> b3=b"""'xyz'"""
>>> print(b3,type(b3))
b'xyz' <class 'bytes'>
>>> print(b3[0],b3[1],b3[2])
120 121 122
```

In addition to the literal forms, bytes objects can be created in a number of other ways:

- A zero-filled bytes object of a specified length: bytes(10)
- From an iterable of integers: bytes(range(20))
- Copying existing binary data via the buffer protocol: bytes(obj)

**Example:**

```
>>> b4=bytes(10)
```

```
>>> print(b4)
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> for a in b4:
    print(a,end=' ')


0 0 0 0 0 0 0 0 0 0
>>> b5=bytes(range(65,91))
>>> print(b5)
b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> for x in b5:
...     print(x,end=' ')
...
...
65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88
89 90
>>> b6=b'\xff'
>>> print(b6)
b'\xff'
>>> print(b6[0])
255
>>> b7=b'\x123'
>>> print(b7[0])
18
```

**bytearray**

bytearray objects are a mutable counterpart to bytes objects.

- Creating an empty instance: bytearray()
- Creating a zero-filled instance with a given length: bytearray(10)
- From an iterable of integers: bytearray(range(20))
- Copying existing binary data via the buffer protocol: bytearray(b'Hi!')

Example

```
ba1=bytearray()
print(ba1,type(ba1))
bytearray(b'') <class 'bytearray'>
ba1.append(65)
>>> ba1.append(66)
>>> ba1.append(67)
>>> print(ba1)
bytearray(b'ABC')
>>> for x in ba1:
...     print(x,end=' ')
...
...
65 66 67
>>> del ba1[0]
>>> print(ba1)
bytearray(b'BC')
>>> ba1[0]=66
>>> print(ba1)
bytearray(b'BC')
>>> ba1[0]=97
>>> print(ba1)
bytearray(b'aC')
>>> ba2=bytearray(10)
>>> print(ba2)
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
>>> ba2=bytearray(range(97,122))
>>> print(ba2)
bytearray(b'abcdefghijklmnopqrstuvwxy')
>>> for x in ba2:
...     print(x,end=' ')
...
...
97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114
115 116 117 118 119 120 121
>>> ba3=bytearray(b'AB')
>>> print(ba3)
bytearray(b'AB')
```

**sets**

sets are unordered collections, insertion order is not preserved.
Sets are non sequence collections, it does not support indexing and slicing.
Sets does not allows duplicate values.

In application development sets are used to group individual values/objects where duplicates are not allowed and to perform mathematical set operations (union, intersection, difference, …) and to remove duplicates from sequences and membership testing.

Sets can be homogeneous (similar) or heterogeneous (different type)

Set allows only immutable objects (hashable objects).
Inside the set data is organized using a data structure called hashing.

Python support 2 set collections

1. Set → Mutable
2. Frozenset → Immutable

**Set**
Set is mutable unordered collection, after creating set changes can be done (adding and removing).

**How to create set?**
Sets can be created by several means:

1. Use a comma-separated list of elements within braces: {'jack', 'sjoerd'}
2. Use a set comprehension: {c for c in 'abracadabra' if c not in 'abc'}
3. Use the type constructor/function : set(), set('foobar'), set(['a', 'b', 'foo'])

**Note:** empty set cannot represent within empty curly braces {}

**How to create empty set in python?**

**Ans:** empty set is created using set() function.

```
>>> A={}
>>> print(A,type(A))
{} <class 'dict'>
>>> A=set()
>>> print(A,type(A))
set() <class 'set'>
```

**Example:**

```
>>> B={10,20,30,40,50}
>>> print(B)
{50, 20, 40, 10, 30}
>>> C={1.5,2.5,3.5,4.5,5.5}
>>> print(C)
{1.5, 2.5, 3.5, 4.5, 5.5}
>>> D={"naresh","ramesh","kishore","kiran"}
>>> print(D)
>>> E={10.0,20.0,30.0,40.5,50.7}
>>> print(E)
{50.7, 20.0, 40.5, 10.0, 30.0}
>>> F={1,2,3,4,5}
>>> print(F)
{1, 2, 3, 4, 5}
>>> G={10,10,10,10,10}
>>> print(G)
{10}
>>> H={10,20,30,10,20,30,10,20,30}
>>> print(H)
{10, 20, 30}
>>> I={10,10.0}
>>> print(I)
{10}
```

**set() function is used to convert other iterables into set type**

**Example**

```
>>> A=set()
>>> print(A)
set()
>>> B=set("JAVA")
>>> print(B)
{'A', 'V', 'J'}
>>> C=set(range(10,110,10))
>>> print(C)
{100, 70, 40, 10, 80, 50, 20, 90, 60, 30}
>>> D=set([1,2,3,4,5,1,2,3,4,5,1,2,3,4,5])
>>> print(D)
{1, 2, 3, 4, 5}
>>> E=set((1,2,3,4,5))
>>> print(E)
{1, 2, 3, 4, 5}
>>> F=set({1,2,3,4,5})
>>> print(F)
{1, 2, 3, 4, 5}
```

**How to read content of set?**

Content of set can be read in 3 ways

1. Using for loop
2. Using iterator object
3. Using enumerate object

**From set data cannot read using index and slicing**

```
>>> A={10,20,30,40,50}
>>> print(A[0])
Traceback (most recent call last):
  File "<pyshell#87>", line 1, in <module>
    print(A[0])
TypeError: 'set' object is not subscriptable
```

```
>>> A[:3]
Traceback (most recent call last):
  File "<pyshell#88>", line 1, in <module>
    A[:3]
TypeError: 'set' object is not subscriptable
```