Partition methods

1. partition
2. rpartition

str.partition(*sep*)
Split the string at the first occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings.

str.rpartition(*sep*)
Split the string at the last occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing two empty strings, followed by the string itself.

```
>>> s1="a,b,c,d,e"
>>> t1=s1.partition(",")
>>> print(s1)
a,b,c,d,e
>>> print(t1)
('a', ',', 'b,c,d,e')
>>> t2=s1.rpartition(",")
>>> print(t2)
('a,b,c,d', ',', 'e')
>>> name="rama rao"
>>> t1=name.partition(" ")
>>> print(t1)
```

str.replace(*old*, *new*[, *count*])
Return a copy of the string with all occurrences of substring *old* replaced by *new*. If the optional argument *count* is given, only the first *count* occurrences are replaced.

```
>>> s2="python java oracle java .net mysql"
>>> s3=s2.replace("java","c++")
>>> print(s2)
python java oracle java .net mysql
>>> print(s3)
```

```
python c++ oracle c++ .net mysql
>>> s4=s2.replace("java","c++",1)
>>> print(s4)
python c++ oracle java .net mysql
>>>
```

str.expandtabs(*tabsize=8*)
Return a copy of the string where all tab characters are replaced by one or more spaces, depending on the current column and the given tab size. Tab positions occur every *tabsize* characters (default is 8, giving tab positions at columns 0, 8, 16 and so on).

```
>>> s3="\t\t"
>>> s4=s3.expandtabs()
>>> print(s4)

>>> print(len(s4))
16
>>> s5="empno\tename\tsalary"
>>> print(s5.expandtabs())
empno   ename   salary
>>> print(s5.expandtabs(16))
empno           ename           salary
```

str.count(*sub*[, *start*[, *end*]])
Return the number of non-overlapping occurrences of substring *sub* in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

```
>>> s1="abcabcabcdefdde"
>>> s1.count("a")
3
>>> s1.count("b")
3
>>> s1.count("ab")
3
>>> s1.count("cd")
1
>>> s1.count("d")
```

```
3
>>> s1.count("a",0,3)
1
```

bytes

Bytes objects are immutable sequences of single bytes. Since many major binary protocols are based on the ASCII text encoding, bytes objects offer several methods that are only valid when working with ASCII compatible data and are closely related to string objects in a variety of other ways.

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.

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:

```
>>> b1=bytes(10)
>>> print(b1)
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> b1[0]
0
>>> b1[-1]
0
>>> for value in b1:
    print(value)


0
0
0
0
0
0
0
0
0
0
>>> b2=bytes(range(65,70))
>>> print(b2)
b'ABCDE'
>>> print(type(b2))
<class 'bytes'>
>>> b3=b'ABCDE'
>>> type(b3)
<class 'bytes'>
>>> print(b3)
b'ABCDE'
>>> for b in b3:
    print(b,end=' ')


65 66 67 68 69
>>> s1='ABCDE'
>>> type(s1)
<class 'str'>
>>> b4=s1.encode()
>>> b4
b'ABCDE'
>>> for x in b4:
    print(x,end=' ')


65 66 67 68 69
>>> for x in s1:
```

```
...     print(x,end=' ')
...
...
A B C D E
>>> list1=[65,67,68,69,70]
>>> b5=bytes(list1)
>>> print(b4)
b'ABCDE'
>>> for x in b5:
...     print(x,end=' ')
...
...
65 67 68 69 70

>>> b6=b'ABCDE'
>>> s2=b6.decode()
>>> print(type(b6))
<class 'bytes'>
>>> print(type(s2))
<class 'str'>
```

encode() is a method of string, which convert string to bytes
decode() is a method of bytes, which convert bytes to string

bytearray
bytearray objects are a mutable counterpart to bytes objects.

bytearray is created with following methods.

- 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!')`

Bytearray is mutable sequence and support all mutable methods

1. Append()
2. Insert()
3. Remove()

4. Extend()
5. Clear()
6. Pop()
7. Del

Example:

```
>>> b1=bytearray()
>>> print(b1)
>>> bytearray(b'')
>>> b1.append(65)
>>> b1.append(66)
>>> print(b1)
bytearray(b'AB')
>>> b1[0]
65
>>> b1[1]
66
>>> del b1[0]
>>> print(b1)
bytearray(b'B')
>>> b2=bytearray(10)
>>> print(b2)
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
>>> b2[0]=65
>>> print(b2)
bytearray(b'A\x00\x00\x00\x00\x00\x00\x00\x00\x00')
>>> b2[-1]=97
>>> print(b2)
bytearray(b'A\x00\x00\x00\x00\x00\x00\x00\x00a')
```

Sequences

list	Mutable
tuple	Immutable
range	Immutable
string	Immutable
bytes	Immutable
bytesarray	Mutable

1. Sequences of index based

2. Sequences ordered collection
3. Sequences allows indexing and slicing
4. Sequences allows duplicates

Sets

Set
frozenset