bytes and bytearray (Buffered Types)

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

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

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

Example:

```
str1="ABC"
for x in str1:
    print(x)
b1=b"ABC"
for x in b1:
    print(x)
b2=b'abc'
for x in b2:
    print(x)
b3=b'''abc'''
for x in b3:
    print(x)

print(type(str1),str1)
print(type(b1),b1)
```

Output

```
A
B
C
65
```

```
66
67
97
98
99
97
98
99
<class 'str'> ABC
<class 'bytes'> b'ABC'
```

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:

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

Example:

```
b1=bytes(10)
print(b1)
for x in b1:
    print(x,end=' ')

print()
b2=bytes(range(65,91))
print(b2)
for x in b2:
    print(x,end=' ')

b3=bytes([97,98,99])
print(b3)
for x in b3:
```

```
print(x,end=' ')
```

Output

```
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
0 0 0 0 0 0 0 0 0 0
b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
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 b'abc'
97 98 99
```

bytearray

bytearray objects are a mutable counterpart to bytes objects. After creating bytearray object changes can be done using mutable operations.

1. append()
2. remove()
3. extend()
4. pop()
5. clear()
6. sort()
7. insert()
8. del keyword

Note: all the operations of list can be applied on bytes object

How to create byte array?

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

Example:

```
b1=bytearray()
```

```
print(b1)
bytearray(b'')
b1.append(65)
print(b1)
bytearray(b'A')
b1.append(66)
>>> print(b1)
bytearray(b'AB')
>>> del b1[0]
>>> print(b1)
bytearray(b'B')
>>> b1[0]=65
>>> print(b1)
bytearray(b'A')
```

Example:

```
>>> b2=bytearray(5)
>>> print(b2)
bytearray(b'\x00\x00\x00\x00\x00')
>>> i=0
>>> for x in range(65,70):
...     b2[i]=x
...     i=i+1
...
...
>>> print(b2)
bytearray(b'ABCDE')
```

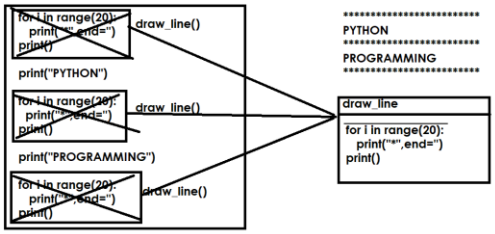
Example:

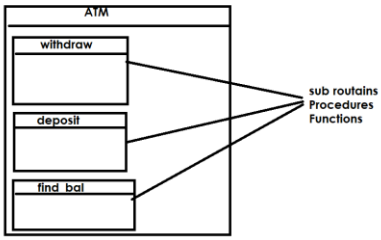
```
>>> b3=bytearray(range(65,70))
>>> print(b3)
bytearray(b'ABCDE')
>>> b4=bytearray(b3)
>>> print(b4)
bytearray(b'ABCDE')
```

Functions

Python is multi paradigm programming language. It allows developing program using different programming paradigms. A programming paradigm defines set of rules and regulations for organizing of data and instructions.

1. Monolithic Programming
2. Procedural Programming
3. Modular Programming
4. Object Oriented Programming
5. Functional Programming





The problem with monolithic programming is,

1. Code redundancy
2. Occupy more space
3. Maintaining code is complex
4. Not easy to understand

Monolithic is organizing everything in one program in sequential order.

To overcome above problems we use procedural oriented programming.

In procedural oriented programming code or instructions are organized by dividing into small pieces and this small pieces are called sub routines (Procedure/Function).

Advantage of POP

1. Modularity: dividing instructions according their operations into small pieces
2. Reusability: Allows to write code once and use many times
3. Readability: Easy to understand
4. Efficiency: It avoids redundant code and increase efficiency of program (less space)

Functions are building blocks of procedural oriented programming.

What is function?

A function is small program within program.
A function is self contained block which contain set of instructions to perform operation
A function is named block

Advantage of Functions

1. Modularity: dividing instructions according their operations into small pieces
2. Reusability: Allows to write code once and use many times
3. Readability: Easy to understand
4. Efficiency: It avoids redundant code and increase efficiency of program (less space)

Functions are 2 types

1. Predefined functions
2. User defined functions

Predefined functions

The existing functions or the functions provided by python or third party vendors are called predefined functions.

Example:

```
>>> len("abc")
3
>>> sorted("cgd")
['c', 'd', 'g']
```

```
>>> print("Hello")
Hello
>>> input()
10
'10'
>>> max(10,20,30)
30
```

Predefined functions are also called library functions.

User defined functions
The functions developed by programmer are called user defined functions. These are application specific functions.
Example: deposit(), login(), logout(), withdraw(), findresult(),…

Basic steps to write function or to work with functions

1. **Define function (OR) Write function**
2. **Invoke function or call the function or using function**

Writing function or defining function

In python function is defined using **"def"** keyword

1. Function name
2. Parameters (optional)
3. Function body (include instructions)

Syntax of writing function:

```
def function-name([parameters]):
    statement-1
    statement-2
```

A function can be defined without parameters
A function can be defined with parameters

Syntax of calling or invoking function
function-name(arguments)

Example:
# Writing Function 	
def sayhello():
  print("Hello Welcome To Functions")
 	
# Call or Use Function 	
sayhello()
sayhello()
sayhello()
sayhello()
sayhello()
sayhello()
	Hello Welcome To Functions
	Hello Welcome To Functions
	Hello Welcome To Functions
	Hello Welcome To Functions
	Hello Welcome To Functions
	Hello Welcome To Functions

Example
 	
def add():
  print("add function")
def sub():
  print("sub function")
def multiply():
  print("multiply function")
def div():
  print("division function")
 	
#main program
 	
div()
add()
div()
sub()
multiply()
	Output
	
	division function
	add function
	division function
	sub function
	multiply function

Memory for function is created or function is loaded in memory when function is invoked or called or executed. Memory is deleted or de-allocated after execution of function.

Whenever function is called or invoked, execution control switched from calling place to called function and after execution of called function return to calling place and continue remaining statements.