How all() works for lists?

```
# all values true
l = [1, 3, 4, 5]
print(all(l))

# all values false
l = [0, False]
print(all(l))

# one false value
l = [1, 3, 4, 0]
print(all(l))

# one true value
l = [0, False, 5]
print(all(l))

# empty iterable
l = []
print(all(l))
```

When you run the program, the output will be:

```
True
False
False
False
True
```

How all() works for strings?

```
s = "This is good"
print(all(s))

# 0 is False
# '0' is True
s = '000'
print(all(s))

s = ''
print(all(s))
```

When you run the program, the output will be:
True
True
True

How all() works with Python dictionaries?

```
s = {0: 'False', 1: 'False'}
print(all(s))

s = {1: 'True', 2: 'True'}
print(all(s))

s = {1: 'True', False: 0}
print(all(s))

s = {}
print(all(s))

# 0 is False
# '0' is True
s = {'0': 'True'}
print(all(s))
```

When you run the program, the output will be:
False
True
False
True
True

How ascii() method works?

```
normalText = 'Python is interesting'
print(ascii(normalText))

otherText = 'Pythön is interesting'
print(ascii(otherText))

print('Pyth\xf6n is interesting')
```

Convert integer to binary using bin()

```
number = 5
print('The binary equivalent of 5 is:', bin(number))
```

When you run the program, the output will be:
The binary equivalent of 5 is: 0b101

Convert an object to binary implementing __index__() method

```
class Quantity:
        apple = 1
        orange = 2
        grapes = 2

        def __index__(self):
        return self.apple + self.orange + self.grapes

print('The binary equivalent of quantity is:', bin(Quantity()))
```

When you run the program, the output will be:
The binary equivalent of quantity is: 0b101

Here, we've sent an object of class Quantity to the bin() method.
The bin() method doesn't raise an error even if the object Quantity is not an integer.
This is because, we have implemented the __index__() method which returns an integer (sum of fruit quantities). This integer is then supplied to the bin() method.

How bool() works?

```
test = []
print(test,'is',bool(test))

test = [0]
print(test,'is',bool(test))

test = 0.0
```

```
print(test,'is',bool(test))

test = None
print(test,'is',bool(test))

test = True
print(test,'is',bool(test))

test = 'Easy string'
print(test,'is',bool(test))
```

When you run the program, the output will be:

```
[] is False
[0] is True
0.0 is False
None is False
True is True
Easy string is True
```

Array of bytes from a string

```
string = "Python is interesting."

# string with encoding 'utf-8'
arr = bytearray(string, 'utf-8')
print(arr)
```

Array of bytes of given integer size

```
size = 5

arr = bytearray(size)
print(arr)
```

Array of bytes from an iterable list

```
rList = [1, 2, 3, 4, 5]

arr = bytearray(rList)
print(arr)
```

How callable() works?

```
x = 5
print(callable(x))

def testFunction():
  print("Test")

y = testFunction
print(callable(y)
```

Callable Object

```
class Foo:
  def __call__(self):
        print('Print Something')

print(callable(Foo))
```

The instance of Foo class appears to be callable (and is callable in this case).

```
class Foo:
  def __call__(self):
        print('Print Something')

InstanceOfFoo = Foo()

# Prints 'Print Something'
InstanceOfFoo()
```

Object Appears to be Callable but isn't callable.

```
class Foo:
  def printLine(self):
        print('Print Something')

print(callable(Foo))
```

The instance of Foo class appears to be callable but it's not callable. The following code will raise an error.

```
class Foo:
  def printLine(self):
        print('Print Something')

print(callable(Foo))

InstanceOfFoo = Foo()
# Raises an Error
# 'Foo' object is not callable
InstanceOfFoo()

# ‘its a comment line comand’
```

Convert string to bytes

```
string = "Python is interesting."

# string with encoding 'utf-8'
arr = bytes(string, 'utf-8')
print(arr)
```

Create a byte of given integer size

```
size = 5

arr = bytes(size)
```

```
print(arr)
```

Convert iterable list to bytes

```
rList = [1, 2, 3, 4, 5]

arr = bytes(rList)
print(arr)
```