

# Network Data and Network Errors

# Bytes and Strings

- ▶ In Python, you will normally represent bytes in one of two ways:
  - ▶ **Integer** whose value happens to be between 0 and 255
  - ▶ **a length-1 byte string** where the byte is the single value that it contains.
  - ▶ You can type a byte-valued number using any of the typical bases supported in Python source code—binary, octal, decimal, and hexadecimal
- ▶ `0b1100010 == 0o142 == 98 == 0x62`

# Python bytes() Function

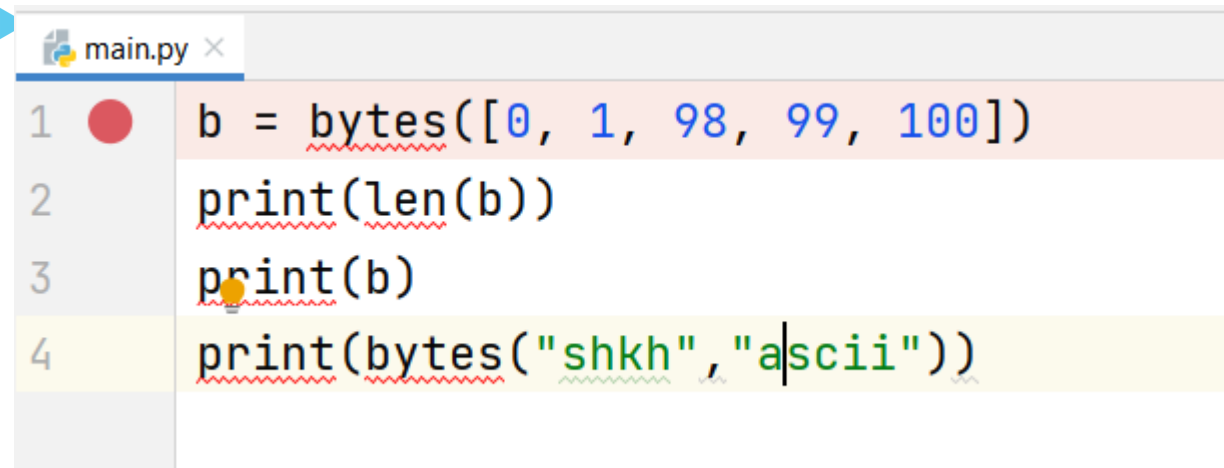
- ▶ The **bytes()** function returns a bytes object.
- ▶ It can convert objects into bytes objects
- ▶ **Syntax**
  - ▶ `bytes(x, encoding, error)`

## Parameter Values

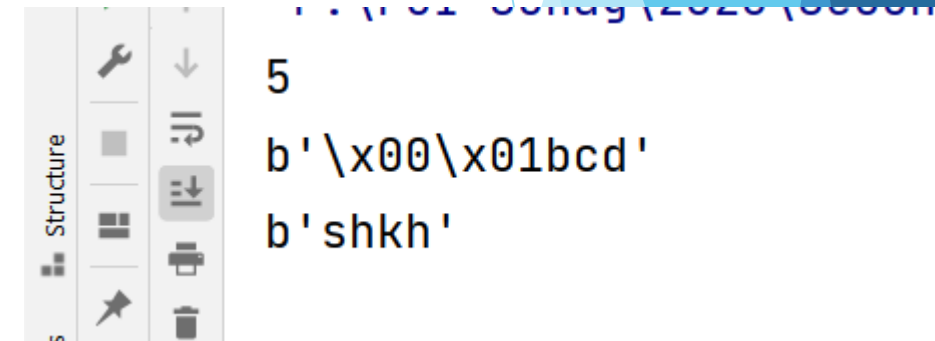
Parameter	Description
<i>x</i>	A source to use when creating the bytes object.  If it is an integer, an empty bytes object of the specified size will be created.  If it is a String, make sure you specify the encoding of the source.
<i>encoding</i>	The encoding of the string
<i>error</i>	Specifies what to do if the encoding fails.

# Bytes and Strings

- ▶ convert a list of such numbers to a byte string by passing them to the bytes()
- ▶ `b = bytes([0, 1, 98, 99, 100])`



```
main.py x
1  b = bytes([0, 1, 98, 99, 100])
2  print(len(b))
3  print(b)
4  print(bytes('shkh', 'ascii'))
```



```
5
b'\x00\x01bcd'
b'shkh'
```

# Character Strings

- ▶ **Encoding** characters means turning a string of real Unicode characters into bytes that can be sent out into the real world outside your Python program.
- ▶ The **encode()** method encodes the string, using the specified encoding. If no encoding is specified, UTF-8 will be used
- ▶ [https://www.w3schools.com/charsets/ref\\_html\\_utf8.asp](https://www.w3schools.com/charsets/ref_html_utf8.asp)

# encode() method

## ► Syntax

► `encode(encoding=encoding, errors=errors)`

### Parameter Values

Parameter	Description												
<i>encoding</i>	Optional. A String specifying the encoding to use. Default is UTF-8												
<i>errors</i>	Optional. A String specifying the error method. Legal values are: <table><tr><td>'backslashreplace'</td><td>- uses a backslash instead of the character that could not be encoded</td></tr><tr><td>'ignore'</td><td>- ignores the characters that cannot be encoded</td></tr><tr><td>'namereplace'</td><td>- replaces the character with a text explaining the character</td></tr><tr><td>'strict'</td><td>- Default, raises an error on failure</td></tr><tr><td>'replace'</td><td>- replaces the character with a questionmark</td></tr><tr><td>'xmlcharrefreplace'</td><td>- replaces the character with an xml character</td></tr></table>	'backslashreplace'	- uses a backslash instead of the character that could not be encoded	'ignore'	- ignores the characters that cannot be encoded	'namereplace'	- replaces the character with a text explaining the character	'strict'	- Default, raises an error on failure	'replace'	- replaces the character with a questionmark	'xmlcharrefreplace'	- replaces the character with an xml character
'backslashreplace'	- uses a backslash instead of the character that could not be encoded												
'ignore'	- ignores the characters that cannot be encoded												
'namereplace'	- replaces the character with a text explaining the character												
'strict'	- Default, raises an error on failure												
'replace'	- replaces the character with a questionmark												
'xmlcharrefreplace'	- replaces the character with an xml character												

# Character Strings

- ▶ **Decoding** byte data means converting a byte string into real characters.
- ▶ **Syntax**
  - ▶ `Str.decode(encoding='UTF-8',errors='strict')`
- ▶ UTF (**Unicode Transformation Format**).
- ▶ The Unicode Standard is implemented in HTML, XML, JavaScript, E-mail, PHP, Databases and in all modern operating systems and browsers.

Charset	Description
UTF-8	A variable-length character encoding (1 to 4 bytes long). UTF-8 is backwards compatible with ASCII and the preferred encoding for e-mail and web pages.
UTF-16	A variable-length character encoding. UTF-16 is used in all major operating systems like Windows, IOS, and Unix.

# struct module

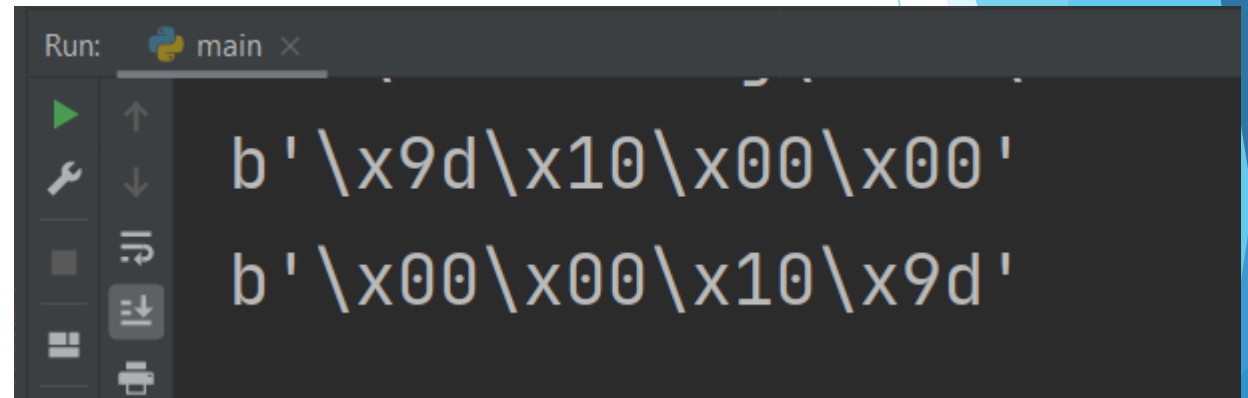
- ▶ **struct module**, provides a variety of operations for converting data to and from popular binary formats.
- ▶ **struct.pack(*format*, *v1*, *v2*, ...)**
  - ▶ Return a bytes object containing the values *v1*, *v2*, ... packed according to the format string *format*.
- ▶ **struct.unpack(*format*, *buffer*)**
  - ▶ which converts the binary data back to Python numbers.



# struct module

```
import struct

print(struct.pack('<i', 4253))
print(struct.pack('>i', 4253))
```



The screenshot shows a terminal window titled 'Run: main'. It displays the output of the Python code. The first line shows the byte sequence for little-endian packing: `b'\x9d\x10\x00\x00'`. The second line shows the byte sequence for big-endian packing: `b'\x00\x00\x10\x9d'`.

**formatting code 'i'**, which uses four bytes to store an integer, and this leaves the two upper bytes zero for a small number like 4253.

# Framing and Quoting

- ▶ **How to delimit your messages so that the receiver can tell where one message ends and the next one begins ??**
  1. The receiver call `recv()` repeatedly until the call finally returns an empty string
  2. Use fixed-length messages
  3. Delimit your messages with special characters.
  4. To prefix each message with its length
  5. Instead of sending just one, try sending several blocks of data that are each prefixed with their length.

# The receiver call `recv()` repeatedly until the call finally returns an empty string

```
message = b''  
while True:  
    more = sc.recv(8192) # arbitrary value of 8k  
    if not more: # socket has closed when recv() returns ''  
        print('Received zero bytes - end of file')  
        break  
    print('Received {} bytes'.format(len(more)))  
    message += more  
print('Message:\n')  
print(message.decode('ascii'))
```

# Server

```
main.py x Server.py x
1 import socket
2 sok=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
3 host=socket.gethostname()
4 port=12345
5 serv_add=(host,port)
6 sok.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
7 sok.bind(serv_add)
8 print('waiting for client')
9 sok.listen()
10 conn,add=sok.accept()
11 conn.shutdown(socket.SHUT_WR)
12 msg=b''
13 while True:
14     data=conn.recv(16)
15     if not data:
16         print('no more data')
17         break
18     print(data)
19     msg=msg+data
20 print(msg.decode())
```

# Client

```
main.py × Server.py × Client.py ×
1  import socket
2  sok=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
3  host=socket.gethostname()
4  port=12345
5  serv_add=(host,port)
6  sok.connect(serv_add)
7  print('Connect to server')
8  sok.shutdown(socket.SHUT_RD)
9  sok.sendall('Client: Hello'.encode())
10 sok.sendall('Client: test send message'.encode())
11 sok.sendall("Client: End".encode())
12 print('data Sent')
13
14
```

# Use fixed-length messages

```
main.py × Server.py × Client.py × recv_len.py ×
1 def recvall(sock, length):
2     data = ''
3     while len(data) < length:
4         more = sock.recv(length - len(data))
5         if not more:
6             raise
7         data += more
8     return data
9
10
```

# How to delimit your messages so that the receiver can tell where one message ends and the next one begins ??

## 3. Delimit your messages with special characters.

- ▶ The receiver would wait in a `recv()` loop until the reply string it was accumulating finally contained the delimiter indicating the end-of-message.

## 4. To prefix each message with its length

- ▶ The length read and decoded, then
- ▶ The receiver can enter a loop and call `recv()` repeatedly until the whole message has arrived (use `recvall(sock, length)`)

Instead of sending just one, try sending several blocks of data that are each prefixed with their length.

► **For read define `get_block(sock)` method**

- **Read length struct size and unpack to convert it to number**

```
header_struct = struct.Struct('!l') # messages up to 2**32 - 1 in length
```

```
len= recvall(sock, header_struct.size)
```

```
(block_length,) = header_struct.unpack(len)
```

- **Read data using `recvall(sock, len)`**

```
recvall(sock, block_length)
```



Instead of sending just one, try sending several blocks of data that are each prefixed with their length.

► To write data define put block

```
import struct
header_struct = struct.Struct('!I')
def put_block(sock, message):
    block_length = len(message)
    sock.send(header_struct.pack(block_length))
    sock.send(message)
```

# Server

```
import socket, struct
header_struct = struct.Struct('!I') # messages up to 2**32 - 1 in length
def server(address):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind(address)
    sock.listen(1)
    print('Listening at', sock.getsockname())
    sc, sockname = sock.accept()
    print('Accepted connection from', sockname)
    sc.shutdown(socket.SHUT_WR)
    while True:
        block = get_block(sc)
        if not block:
            break
        print('Block says:', repr(block))
    sc.close()
    sock.close()
```

# Get\_block method

```
def get_block(sock):  
    data = sock.recv(header_struct.size)  
    (block_length,) = header_struct.unpack(data)  
    return recvall(sock, block_length)  
  
def recvall(sock, length):  
    data = ''  
    while len(data) < length:  
        more = sock.recv(length - len(data))  
        print(data)  
        if not more:  
            raise  
        data += more.decode()  
    return data  
server(('127.0.0.1', 12346))
```

# Client

```
main.py × Server.py × Client.py × Server2.py × Client2.py × recv_len.py ×
1  import socket, struct
2  header_struct = struct.Struct('!I')
3  def client(address):
4      sock = socket.socket(socket.AF_INET, socket.SOCK_STR
5      sock.connect(address)
6      sock.shutdown(socket.SHUT_RD)
7      put_block(sock, b'Beautiful is better than ugly.')
8      put_block(sock, b'Explicit is better than implicit.'
9      put_block(sock, b'Simple is better than complex.')
10     put_block(sock, b'')
11     sock.close()
12
13  def put_block(sock, message):
14      block_length = len(message)
15      sock.send(header_struct.pack(block_length))
16      sock.send(message)
17
18  client(('127.0.0.1', 12346))
19
```

# repr() method

- ▶ **repr()**
- ▶ The repr() function returns a printable representation of the given object.

The background features abstract, overlapping geometric shapes in various shades of blue, ranging from light sky blue to deep navy blue. These shapes are primarily located on the right side of the slide, creating a modern, dynamic feel. The rest of the slide has a plain white background.

# Thank You