# Lab: External representation of data

Bits and bytes are the basic units of data underlying all the applications and programs. This is usually kept hidden, however with network programming, because we have to pass bytes through sockets and put them on the wire, we have to deal with them directly and explicitly, instead of being hidden underneath layers of data structures.

Suppose you want to transmit a string of characters through a socket, you will need to encode the string as a sequence of bytes. *Encoding* means converting a string of characters to a sequence of bytes (byte string). *Decoding* means converting a byte string back to a string of characters. When data is going out to the network through a socket, we encode it to bytes. On the receiving end

An example is the ASCII encoding, which assigns to each character an integer from 0 to 127, requiring 7 bits. The first 32 integers (byte values) are for control commands. The remaining 96 integers (byte values) represent various characters and symbols. You can see all of them by typing in the interactive Python shell:

```
>>> print(' '.join(chr(i) for i in range(32, 128)))
```

There are extensions to the ASCII code that uses the (highest) 128 byte values for region-specific characters, e.g., letters with accents. In these, each character of the string is always represented by a single byte.

```
>>> 'réseau'.encode('ascii')
```

will give an error that ASCII cannot encode the second letter because of its accent. But an extension called *latin1* of ASCII will be able to encode it:

```
>>> 'réseau'.encode('latin1')
```

The above will give the bytes that represent this word.

Python 3 actually uses the more recent **Unicode** character encoding, which can represent a lot more than 128 different symbols, and of course uses more than one byte per symbol. There are several variants of the Unicode encoding, e.g., UTF-8, UTF-32.

The popular UTF-8 encoding keeps it compact by varying the number of bytes used for each symbol, which can complicate things. The UTF-32 encoding avoids such complication by using a fixed 32+4=36 bytes for each symbol, paying with the use of more bytes. You can compare:

```
>>> 'réseau'.encode('utf-8')
>>> len(_)
>>> 'réseau'.encode('utf-32')
>>> len(_)
```

UTF-8 uses 7 bytes while UTF-32 uses 28.

You can get an error trying to decode a byte string that was encoded by a different encoding, or trying to encode symbols that cannot by represented by the specified encoding. Both of the below will give errors.

```
>>> ('réseau'.encode('utf-8')).decode('ascii')
>>> 'réseau'.encode('ascii')
```

To not get errors, you can either tell the encoding to replace the un-encodable symbol with one that is encodable, or to ignore it and remove it from the string. Try this out for yourself:

```
>>> ('réseau'.encode('utf-8')).decode('ascii', 'replace')
>>> ('réseau'.encode('utf-8')).decode('ascii', 'ignore')
>>> 'réseau'.encode('ascii', 'replace')
>>> 'réseau'.encode('ascii', 'ignore')
```

## Framing messages

There are several ways client and server may exchange messages. The client could transmit a stream of bytes in one shot and then close the connection, all the server needs to do is receive data until the client closes their socket.

Alternatively, the client could send multiple messages to the server, one after another. In this case, the server must be able to tell the boundaries of these messages. You can choose to use special delimiting characters to mark the boundaries of each message. This is called *framing* the messages. The choice must be careful, it must be a character that cannot possibly appear in the message content. Sometimes that is not possible, if the messages can contain any character or symbol.

Another solution is to prefix each message with the length of the message in bytes, that way, the server knows exactly how many bytes to receive for each message. We do this in the example program, `tcp_chunks.py`, where the client sends three messages or chunks to the server, then sends an empty byte string to indicate the end of communication.

To execute, first run the server (you may choose a port number different from 1234, as long as both server and client use the same one):

```
python tcp_chunks.py server 0.0.0.0 1234
```

Then run the client (assuming the IP address of server is 10.0.2.15):

```
python tcp_chunks.py client 10.0.2.15 1234
```

There are several things to note in this example program.

1. Python 3 provides a bytes object, which is a byte string (default is ASCII encoding) that can be transmitted directly through a socket. To convert a string to a bytes object, simply prepend the string with `b`, e.g., `b'Hello World'` is actually a byte string. The bytes object is an array of integers, each integer represents the encoding of a character or symbol.

2. The `struct` module is imported for converting between C structs and Python bytes objects. A header struct is created to hold the length of the message (number of bytes). By calling the `pack()` method of the header struct and passing to it the length of each message chunk, the length is converted to a bytes object that can be transmitted to the server. The header struct is created by the line
   `header = struct.Struct('!I')`
   The symbol `I` means an unsigned integer, 32 bits, so the length of a message can be up to $2^{32} - 1$ bytes. The `!` means that the big-endian byte order is used for the integer.

3. When the client sends the three (non-empty) message chunks, it's done in the function `send_chunk()`. It first sends the header as a bytes object, then sends the actual message chunk, also a bytes object since the message strings are prepended by `b`.

4. The server receives messages by calling the function `recv_chunk()`. It first receives the the number of bytes that the header struct uses, which is a fixed value. Then it converts this bytes object to its original struct, by calling `header.unpack(data)`. The struct contains, in this case, only the length of the subsequent message.

   Now the server knows the length or number of bytes of the message it will be receiving, passes it to the `recvall()` function, which receives the actual message.

   In `recvall()`, although unlikely, the message could be received in multiple packets, and if that occurs, they are all joined into a single string and returned.

## Task 1: client user input

As an exercise, you could modify the client code a little. One option is to have the client prompt for user input from command-line for sentences. Each sentence is converted to a bytes object and sent to the server. Then client prompts for another one, until the user inputs an empty string, resulting in the empty message being sent to the server which shuts down the server (same as before). The client should also quit upon getting the empty string.

A second option is to have the client read line by line from a text file and send these lines one at a time to the server. Both client and server should quit after the whole file has been sent.

Note that a string in variable `str` can be converted to a Python bytes object while specifying a particular encoding: `bytes(str, 'utf-8')`

To read from command-line, the bulit-in function `input()` can be used:
`sentence = input("Enter a sentence:  ")`

To read lines one at a time from a file, first open the file and get the file object `f`, then:
`for line in f:  ...`

You can do either or both of these options.

## Native data serialization: Pickle

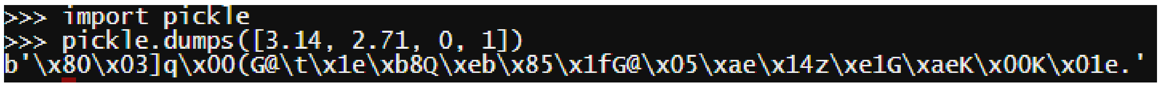
So far we've only seen string data being sent, however in general, you may want to send any arbitrary data structure, e.g., list of numbers, list of strings, array of arrays, maps, etc. These more complicated data structures need to be essentially 'flattened' and put into a flat sequence which can then be transmitted.

The process of converting data structures to a linear sequence of characters or bytes is called ***serialization***.

The opposite process of converting back is called ***deserialization***.

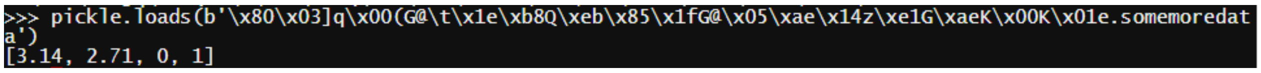
When we transmit and receive data over the network, we need to serialize data structures into byte strings (e.g., bytes object in Python) that is ready to be transmitted over the network. In the above, we have basically done this serialization ourselves. There are existing libraries that provide serialization of data.

The Python native library for serialization is `pickle`. The `dumps()` and `dump()` methods convert the given contents of a Python data structure to a bytes object. The `load()` and `loads()` reverses the process. For example, you can try serializing a list of numbers (i.e., converting it to a byte string):



```
>>> import pickle
>>> pickle.dumps([3.14, 2.71, 0, 1])
b'\x80\x03]q\x00(G@\t\x1e\xb8Q\xeb\x85\x1fG@\x05\xae\x14z\xe1G\xaeK\x00K\x01e.'
```

Notice the dot at the end of the binary string, that is the delimiter that `pickle` uses. So when the pickle is loaded, only data up to that dot is read. Suppose we add some more bytes after the dot in the pickle above, when it is loaded, only the data up to the dot will be read, as desired.



```
>>> pickle.loads(b'\x80\x03]q\x00(G@\t\x1e\xb8Q\xeb\x85\x1fG@\x05\xae\x14z\xe1G\xaeK\x00K\x01e.somemoredat
a')
[3.14, 2.71, 0, 1]
```

An analogous pair of methods `dump()` and `load()` allows pickles to be written to a file and read from a file. This is useful to us because we already know that a socket can be made into a file object. If both server and client made their sockets file objects, then they can read and write pickles via these files, which results in them exchanging pickles over the network. This way, the pickle library is doing all the work of the delimiting of messages for us.

### Task 2: sockets as file objects

Modify both the server and client code in `tcp_chunks.py` to have the client create a pickle for each sentence it needs to send, create the file object for its socket, then write the pickles into that file. Similarly, modify the server to create a file for its socket and read pickles from that file, then print out the original strings from those. You'll find that the program will become much shorter.

## To Submit

Submit Task 1 and 2.