**Table of Contents**

1. Introduction

2. BinaryEncoder Class

- 2.1 Constructor (‘\_\_init\_\_’)

- 2.2 Method: ‘encode\_integer’

- 2.3 Method: ‘encode\_float’

- 2.4 Method: ‘encode\_string’

- 2.5 Method: ‘get\_buffer’

3. BinaryDecoder Class

- 3.1 Constructor (‘\_\_init\_\_’)

- 3.2 Method: ‘decode\_integer’

- 3.3 Method: ‘decode\_float’

- 3.4 Method: ‘decode\_string’

4. Test Encoding and Decoding

- 4.1 Purpose

- 4.2 Performance Testing

- 4.3 Assertions and Data Integrity

5. Example Usage

6. Key Points

7. Potential Improvements

**Introduction**

This script offers a Binary Encoder and Decoder for integers, floating-point numbers, and strings. It utilizes Python's ‘struct’ module to encode and decode data with a defined format, big-endian. The script efficiently handles data encoding using a ‘bytearray’ and ensures integrity through simple assertions.

The main components of this script are:

1. BinaryEncoder: A class to encode data into binary format.

2. BinaryDecoder: A class to decode binary data back into its original form.

3. Test Encoding and Decoding: A function to verify encoding and decoding, as well as measure the performance of the process.

**BinaryEncoder Class**

The ‘BinaryEncoder’ class is responsible for converting different data types (integers, floats, and strings) into binary format.

2.1 Constructor (‘\_\_init\_\_’)

def \_\_init\_\_(self):

self.buffer = bytearray() # Use bytearray for memory-efficient binary storage

- Purpose: Initializes a buffer to hold the binary data.

- Explanation: A ‘bytearray’ is used to store the binary data, which is mutable and allows for efficient appending of new binary data without reallocating memory every time.

2.2 Method: ‘encode\_integer’

def encode\_integer(self, value: int) -> bytes:

“““Encodes an integer into binary format.”””

binary\_data = struct.pack(‘!I’, value) # ‘!I’ -> Big-endian unsigned integer

self.buffer.extend(binary\_data) # Add to the buffer

return binary\_data

- Purpose: Converts an integer (‘int’) to its binary representation.

- Explanation:

- ‘struct.pack(‘!I’, value)’ converts the integer to a binary format.

- The ‘!I’ format character indicates that the data must be encoded as a “big-endian unsigned integer” (4 bytes).

- The method adds this binary data to the `bytearray` buffer.

2.3 Method: ‘encode\_float’

def encode\_float(self, value: float) -> bytes:

“““Encodes a float into binary format.”””

binary\_data = struct.pack(‘!f’, value) # ‘!f’ -> Big-endian float

self.buffer.extend(binary\_data)

return binary\_data

- Functionality: Converts a floating-point number (‘float’) into binary format.

- Description:

- ‘struct.pack(‘!f’, value)’ converts the floating-point value to a binary representation.

- The ‘!f’ format character indicates that the value is a “big-endian float” (4 bytes).

- The binary data is appended to the ‘buffer’.

2.4 Method: ‘encode\_string’

def encode\_string(self, value: str) -> bytes:

“Encodes a string into binary format.”

encoded\_str = value.encode('utf-8') # Convert string to bytes (UTF-8 encoding)

length = len(encoded\_str) # Store the length of the string

binary\_data = struct.pack('!I', length) + encoded\_str # Prepend length of the string

self.buffer.extend(binary\_data)

return binary\_data

- Functionality: It takes a string ‘str’ and returns its binary representation, encoding the length of the string.

- Explanation

- First, the string is encoded to UTF-8 bytes using ‘value.encode(‘utf-8’)’.

- The length of the string is encoded as a 4-byte unsigned integer (‘!I’).

- The method combines the length and the UTF-8 encoded string into one binary stream and appends it to the buffer.

2.5 Method: ‘get\_buffer’

def get\_buffer(self):

“““Return the final binary buffer.”””

return bytes(self.buffer)

- Description: Returns the final accumulated binary data.

- Explanation: The method converts the mutable ‘bytearray’ into an immutable ‘bytes’ object before returning it.

**BinaryDecoder Class**

The `BinaryDecoder` class is in charge of converting binary data back into their original forms: integers, floats, and strings.

3.1 Constructor (‘\_\_init\_\_’)

self.buffer = binary\_data

self.offset = 0 # Pointer to track current position in the binary data

- Purpose: Initializes the decoder with the binary data and an offset to track the current reading position.

- Explanation: The `buffer` stores the binary data, and the ‘offset’ keeps track of where we are while decoding the data.

3.2 Method: ‘decode\_integer’

def decode\_integer(self) -> int:

“““Decodes an integer from binary format.”””

value = struct.unpack\_from(‘!I’, self.buffer, self.offset)[0]

self.offset += 4 # Advance the offset by 4 bytes; this is the size of an unsigned int

return value

- ‘struct.unpack\_from(‘!I’, self.buffer, self.offset)’ reads the 4-byte integer starting from the current ‘offset’ in the binary data.

- The offset is advanced by 4 bytes (the size of an unsigned integer) after the decoding.

3.3 Method: ‘decode\_float’

def decode\_float(self) -> float:

“““Decodes a float from binary format.”””

value = struct.unpack\_from(‘!f’, self.buffer, self.offset)[0]

self.offset += 4 # Move the offset forward by 4 bytes (size of a float)

return value

- Purpose: Decodes a 4-byte floating-point number from the binary data.

- Explanation:

- Similar to the integer decoding, but for floating-point numbers.

- The ‘offset’ is updated by 4 bytes.

3.4 Method: ‘decode\_string’

def decode\_string(self) -> str:

“Decodes a string from binary format.”

length = struct.unpack\_from(‘!I’, self.buffer, self.offset)[0] # Get the string length

self.offset += 4 # Move the offset forward by 4 bytes for the length

value = self.buffer[self.offset:self.offset + length].decode(‘utf-8’) # Decode the string

self.offset += length # Move the offset forward by the length of the string

return value

- Purpose: Decodes a string from binary format.

- Explanation:

- The string’s length is first decoded as an unsigned integer.

- The method then extracts the correct number of bytes corresponding to the string and decodes it using UTF-8.

- The ‘offset’ is updated after decoding the string.

**Test Encoding and Decoding**

4.1 Purpose

The ‘test\_encoding\_decoding’ function checks that encoding and decoding are working correctly.

4.2 Performance Testing

This function measures the time it takes to encode and decode:

start\_time = time.time()

# Encoding

encoding\_time = time.time() - start\_time

start\_time = time.time()

# Decoding

decoding\_time = time.time() - start\_time

4.3 Assertions and Data Integrity

Assertions are used to test the integrity of the encoded and decoded data:

assert integer\_data == decoded\_integer

assert abs(float\_data - decoded\_float) < 1e-6

assert string\_data == decoded\_string

These will check that the decoded data matches the original data.

**Example Usage**

if \_\_name\_\_ == “\_\_main\_\_”:

test\_encoding\_decoding()

The ‘test\_encoding\_decoding’ function is called to test the encoding and decoding functionality.

**Key Points**

- Big-endian Format: Data is encoded in the big-endian format (‘!’), which is necessary for data transfer between systems of different byte orders.

- Efficient Memory Usage: ‘bytearray’ is used for efficient memory management when appending binary data.

- String Encoding: Strings are encoded in UTF-8 format, so the system can be compatible with international characters.

**Possible Improvements**

- Error Handling: Error handling for malformed binary data during decoding.

- Custom Formats: Support additional formats like signed integers or different floating-point types.

- Optimized String Handling: Optimize string encoding/decoding for very large strings by using compression or chunking techniques.