CS 115 Lab 7: Data Encodings

This lab was written as a collaboration with Jamil, Rohan, Sydney, and the instructor.

We have now seen in class computers represent all data as bits, which can be organized into bytes. **1 byte is 8 bits**. In this lab, you will explore more about how Python represents numbers and text in a machine-readable format.

Encryption and Ciphers

Suppose CA Bob and CA Alice are talking to each other. In typical CA fashion, they both speak as a tuple of strings, and sparsely use punctuation to avoid sending more data than needed. If we send Bob a message with `send_bob()` and Alice a message with `send_alice()`, then their conversation would look something like this:

```
>>> send_bob(("Hey","Bob", "It's", "Alice"))
("Hey","Bob", "It's", "Alice")
>>> send_alice(("Hey","Alice"))
("Hey","Alice")
```

However, Evil Eve wants to see what Bob and Alice are talking about to get ahead. Since Bob and Alice don't encrypt their message, Eve can read their message at will:

```
>>> read_message()
("Hey","Bob", "It's", "Alice")
```

To fix this problem, CA Bob and CA Alice come up with a cipher method to encrypt their communications. They decide to use a Caesar Cipher, a type of cipher that offsets, or shifts, every letter by some amount k. For example, when k = 3, the letter 'A' becomes 'D', 'B' becomes 'E', and so on.

```
>>> caesar_cipher(("abc",),1) #Offset by 1
('bcd',)
>>> caesar_cipher(("abc",),2)#Offset by 2
('cde',)
>>> caesar_cipher(("Hey","Alice", "Send", "The", "Answers", "Key"),3) #Offset by 3
('Kh|', 'Dolfh', 'Vhqg', 'Wkh', 'Dqvzhuv', 'Nh|')
```

Evil Eve can see the encrypted message, but won't be able to understand it. However, since Alice and Bob both know that the offset is 3, they can decrypt messages to each other.

```
>>> decrypt_message(('Kh|', 'Dolfh', 'Vhqg', 'Wkh', 'Dqvzhuv', 'Nh|'), 3)
("Hey","Alice", "Send", "The", "Answers", "Key")
```

But Eve knows that there are only so many combinations to the Caesar Cipher, and if she tries all the possible offsets, she can find the original message.

```
>>> decrypt_caesar_cipher(('Kh|', 'Dolfh', 'Vhqg', 'Wkh', 'Dqvzhuv', 'Nh|'), 1)
('Jg{', 'Cnkeg', 'Ugpf', 'Vjg', 'Cpuygtu', 'Mg{')
>>> decrypt_caesar_cipher(('Kh|', 'Dolfh', 'Vhqg', 'Wkh', 'Dqvzhuv', 'Nh|'), 2)
('Ifz', 'Bmjdf', 'Tfoe', 'Uif', 'Botxfst', 'Lfz')
>>> decrypt_caesar_cipher(('Kh|', 'Dolfh', 'Vhqg', 'Wkh', 'Dqvzhuv', 'Nh|'), 3)
("Hey","Alice", "Send", "The", "Answers", "Key")
```

Now Bob and Alice's communications are exposed. To fix this, they come up with a new type of cipher – The LaGrassian Cipher!

Task 1: LaGrassian Cipher

The LaGrassian Cipher is a variation of the Caesar Cipher. Instead of using a static offset (shift), it changes the offset based on the numerical value of the character you are encrypting. If the character's UTF-8 value (UTF-8 is a library similar to ASCII that encodes characters into numbers) is even, then we add the offset. If not, then we subtract the offset.

```
>>> lagrassian_cipher(("b",), 1) #Moves up one as b = 98
('c',)
>>> lagrassian_cipher(("e",), 1)#Moves down one as e = 101
('d',)
>>> lagrassian_cipher(("bcd",), 1)
('cbe',)
>>> lagrassian_cipher(("bcd",), 5)
('g^i',)
>>> lagrassian_cipher(("I", "am", "so", "cool"), 19)
('6', 'NZ', '`\\', 'P\\\\\x7f')
>>> lagrassian_cipher(('6', 'NZ', '`\\', 'P\\\\\x7f'), 19)
('I', 'am', 'so', 'cool')
```

Implement the LaGrassian Cipher function in `lab7.py`. Note that the offset must be an **odd** number that is less than 65, since 65 is the highest UTF-8 code representing a character ('A').

0.1 Data encoding and format in Python

There is no task for this section, but you will need this reference for the rest of the tasks. In this task, we will inspect how data can be represented as bytes. The `struct` module in Python allows us to do this.

We provide two helper functions to help you inspect data using only the functionality we have learned in class. You are welcome to try out the struct library directly once you have completed the lab.

- `pack_data(data, byte_order, number_format)` where the byte order is `"little endian"`, `"big endian"`, `"native"` or `"network"`
- `print_bytes_as_bits` prints bits grouped by bytes

We encourage you to play with this function by trying different numbers and formats and seeing how they compare to what you expect.

Task 2: Finding the Endianness of your computer

In class, we have assumed numbers are read left to right, from the most significant digit to the least significant digit. In reality, this choice is arbitrary. A **big endian** system puts the *most* significant digit **first**. A **little endian** system puts the *least* significant digit **first**.

Figure out if your system is natively big endian or little endian by:

1. Encoding an integer between 1 and 100 of your choice by hand; let's call it `my_number`. Put the most significant bit on the left (big endian) as shown in class.
2. Call `pack_data(my_number, 'native', 'int')` to get the representation in bytes.
3. Print the representation using `print_bytes_as_bits` to read the encoding.
4. Compare your encoding to what you see in `print_bytes_as_bits`

Report question

1. What bits were printed?
2. Is your system natively big-endian or little-endian? Describe how you figured that out.
3. Instead of using the `native` byte order, use the `network` byte order and answer the same questions from above.

CA Check-In

1. Ask the CA for a number between 1 and 100.
2. Encode it into 2 bytes. Explain the steps you took to reach your result.
3. Compute the 2's complement of that number. Explain the steps you took to reach your result.

Task 3: Two's complement in Python

Now that we know the endianness of our system, we are going to check if Python represents negative numbers using two's complement.

Report question

1. Write an 8-bit representation of a number between -100 and 100 of your choice.
2. Write the negative of that number using 2's complement encoding.
3. Use `pack_data` to get the binary representation of that number as an `int`.
4. Compare the *negative* number you computed by hand to the result of packing the negative number as an `int`.

Optional: try packing numbers using some of these types and inspecting the bytes (`int`, `unsigned int`, `short`, `bool`, `char`, `long`, `float`, `double`). Note that not all data can be encoded with all types. Log anything you noticed that you found intereseting.

Task 4: Floating point numbers

In this task, we will inspect the encoding of floating point numbers. For this section, use the byte order `"network"` `byte_order` as the second argument to `pack_data`.

Report question

1. Compare the result of packing `0.25` to `-0.25` as a ``` ``double'' ``` as the number format. In particular, compare the sign bit(s), exponent (e) and mantissa (m).
2. Compare using ``` ``double'' ``` to ``` ``float'' ``` as the number format: how many **bits** are in each?
3. Python's floating point numbers (including `float`!) is actually a ``` ``double'' ``` as described by (most) other languages! Why would the Python designers choose to use 64 bits to represent fractional numbers?

Bonus: Inspect other numbers to see if Python follows the IEEE 754 standard.

Rubric

Category	Description	Points
Name	For stating your name on every file for submission.	1
Pledge	For writing the full Stevens Honor Pledge on every file for submission.	1
Collaboration	For listing your collaborators.	1
Check-In	CA Check-In during the lab.	10
Task 1:	Correct implementation of `lagrassian_cipher`	30
Task 2:	Process to find endianness of *system*	7
Task 2:	Process to find endianness of *network*	5
Task 3:	Use and interpretation of packing negative number as `int`	15
Task 4:	Interpretation of binary representations of 0.25 and -0.25 packed as a ‘‘`double`’’	10
Task 4:	Reported number of bits using ‘‘`double`’’ and ‘‘`float`’’ number formats	10
Task 4:	Reasonable justification for why Python would use ‘‘`double`’’ instead of ‘‘`float`’’ by default	10
Total		100

Submission Instructions

For this lab/homework, you must submit the following files, with exactly the following names:

- `lab7.py`
- `lab7_report.txt`

Remember to put your name and the Stevens Honor Pledge on every file you submit! Failure to put the pledge can result in marks being deducted, with repeat penalties potentially getting stricter between assignments.

Submissions must be handled through Gradescope (accessible through Canvas). Grades will be released through Gradescope once the lab is graded.

Hints and Suggestions

- For more information about built-in data types in Python, see the first part of `docs.python.org/3/library/stdtypes.html`
- An inline if statement can be made inside a lambda function as such

```
>>> tuple(map(lambda x: "G" if x > 5 else "L", (1, 7, 3)))
('L', 'G', 'L')
```

Notes and Reminders

- **Remember to include docstrings!** If your submission does not have docstrings, or has docstrings that are not related to your code, you will not get credit for them.
- We have provided a test file for you to test your solutions. By running this test file, you should be able to verify that your files are named properly, and that your code works correctly.
- Test frequently! Try out many different cases to verify that your code runs. Our provided test file may not be comprehensive, and so you should ensure to test your code on many other cases as well.

- Code that doesn't compile (throws a `SyntaxError` on being run) will receive a 0 for the assignment! Make sure you submit code that can be run, even if it does not pass all the test cases.
- **Don't forget to add your name and pledge!**