Lecture 20

File IO and Exceptions Review

22 Germinal, Year CCXXX

Song of the day: ***A Horde of One*** *by Disasterpeace (2022).*

Review: Telegram Parsing

Background

Telegrams reached their peak popularity in the 1920s and 1930s when it was cheaper to send a telegram than to place a long-distance telephone call. People would save money by using the word “STOP” instead of periods to end sentences because punctuation was extra while the four character word was free[(1)] . Oftentimes—especially with important communication—these telegrams would be encoded with some sort of cypher. For this program, we're going to build a program that reads an encoded telegram, decode it, and write a new file with the telegram's decoded contents. The word "STOP", in this case, will not only represent a period, but **it will also represent the start of a new line**.

The `get_decoded_lines()` *function*

Start by writing a function, `get_decoded_lines()` , that will accept two parameters: a string containing an encoded message, and an integer representing the decoding key.

You can assume that this string will be valid, and that it will be an encoded telegram message. Therefore, each sentence will be separated by the word "STOP". Using this knowledge, your function must:

1. Decode the whole string **using the imported** `decrypt_string()` **function**, which itself accepts an encoded string and a key integer as parameters. Do *not* write your own Caesar string decoder (unless you really want to, I guess).
2. Return a list of each of the separate lines included in the decoded string, while dispensing of the "STOP" separator.

For example, the following script:

```
def main():
    # The message in encoded_telegram.txt
    sample_string = "Tumyjj IJEF Te dej qbuhj Secijesa je oekh fhuiudsu IJEF Mxqj
    sample_key = 42

    decoded_lines = get_decoded_lines(sample_string, sample_key)
    print(decoded_lines)

main()
```

Will output the following list:

```
['Dewitt', 'Do not alert Comstock to your presence', 'Whatever you do do not pick
```

The `create_decoded_telegram()` *function*

Once you have `get_decoded_lines()` implemented, we can use it to create our decoded telegram.

Write a second function `create_decoded_telegram()` that will accept two string parameters. Both parameters represent file paths—the first of the encoded telegram, and the second containing the name you wish to give your decoded telegram. Both must be `txt` files. You may not assume that the encoded file exists. If the user chooses not to decide on the name of the decoded telegram file, your function must default to the string `"decoded_telegram.txt"` as its second parameter.

The encoded file will *always* be of the following format

```
encryption key
encoded message
```

That is, the first line will contain the key with which the message was encoded, and the second line will contain the message itself. You can assume that the encryption key line will always represent a valid integer.

Optional: If you want a bit more of a challenge, try assuming that the first line might *not* represent a valid integer. If this happens, do *not* create an output file, print an error message, and exit the function.

When the program is fully implemented, it will function as follows:

- Encoded telegram (`encoded_telegram.txt`):

```
42
Tumyjj IJEF Te dej qbuhj Secijesa je oekh fhuiudsu IJEF Mxqjuluh oek te te dej fy
```

- Python script:

```
def main():
    create_decoded_telegram("encoded_telegram.txt", "output.txt")

main()
```

- Decoded telegram (`output.txt` , in case that `encoded_telegram.txt` was opened successfully and had valid contents):

```
Dewitt.
Do not alert Comstock to your presence.
Whatever you do do not pick number 77.
Lutece
```

Review Solution

The first stop in writing any function must be to implement its "skeleton" correct. In the case of `get_decoded_lines()` , we were told that it accepts *two* parameters. I chose to name mine `encoded_message` and `key` , but you could have used any other names as long as they made sense (i.e. `encryption_key` instead of `key` , etc.:

```
def get_decoded_lines(encoded_message, key):
    # function definition
```

We were then told to use `decrypt_string()` , a function that was already written and imported for us. Using `decrypt_string()` , we can get the decrypted version of our entire message:

```
from ceasar_decryptor import decrypt_string

def get_decoded_lines(encoded_message, key):
    decoded_message = decrypt_string(encoded_message, key)
```

We were told that this message will contain multiple sentences separated by the word "STOP", so we must use the `split()` method to isolate them as members of a string. The tick here was to recognise that the separator was not simply the string `"STOP"`, but the string `" STOP "` (notice the two spaces before and after `STOP`). This would get rid of the white space before and after each sentence (indexing through each element with a `for`-loop and using `strip()` was another option).

Finally, return your list:

```
from ceasar_decryptor import decrypt_string

def get_decoded_lines(encoded_message, key):
    decoded_message = decrypt_string(encoded_message, key)
    decoded_message = decoded_message.split(" STOP ")

    return decoded_message
```

Next up was `create_decoded_telegram()` which, as we were told, accepted two parameters representing filepaths. Moreover, we know that the second parameter has the default value `"decoded_telegram.txt"`. Therefore, our function skeleton will look as follows:

```
def create_decoded_telegram(encoded_filepath, decoded_filepath="decoded_telegram.t
    # Function definition
```

Again, your parameter names could have been different to mind.

Since we can't assume that the encoded file exists, we must wrap it inside a `try - except` block to safely attempt to open it first. If it fails to open the file, we can give the user a message explaining the error and exit the function immediately using `return`:

```
def create_decoded_telegram(encoded_filepath, decoded_filepath="decoded_telegram.t
    try:
        input_file = open(encoded_filepath, 'r')
    except FileNotFoundError:
        print("NON-FATAL ERROR: Encoded telegram file {} could not be found.".forr
        return
```

Provided that the file *does* exist, we can proceed at extracting our two pieces of data. The first line is the encryption key. If you assumed that this line will always be a valid integer, you would simply do this:

```
    key = int(input_file.readline().strip())
```

If you didn't, we would have to wrap our casting attempt in a `try - except` block, and `return` if things didn't work out:

```
    key = input_file.readline().strip()

    try:
        key = int(key)
    except ValueError:
        print("NON-FATAL ERROR: Invalid encryption key '{}'. Must be able to be caste
        return
```

I'll keep the second version in the full solution file, but both ways were allowed in this problem:

```
    def create_decoded_telegram(encoded_filepath, decoded_filepath="decoded_telegram.t
        try:
            input_file = open(encoded_filepath, 'r')
        except FileNotFoundError:
            print("NON-FATAL ERROR: Encoded telegram file {} could not be found.".form
            return

        key = input_file.readline().strip()

        try:
            key = int(key)
        except ValueError:
            print("NON-FATAL ERROR: Invalid encryption key '{}'. Must be able to be ca
            return
```

We can then extract the actual message using another call to `readline().strip()`. Since we are done with `input_file`, you could have closed it here, although closing it at the end of the function is fine as well:

```
def create_decoded_telegram(encoded_filepath, decoded_filepath="decoded_telegram.tx
    try:
        input_file = open(encoded_filepath, 'r')
    except FileNotFoundError:
        print("NON-FATAL ERROR: Encoded telegram file {} could not be found.".forma
        return

    key = input_file.readline().strip()

    try:
```

```
        key = int(key)
    except ValueError:
        print("NON-FATAL ERROR: Invalid encryption key '{}'. Must be able to be cas
        return

    encoded_message = input_file.readline().strip()
    input_file.close()
```

So now we have both pieces of information, our encrypted message and our encryption key, necessary to use our `get_decoded_lines()` function:

```
decoded_lines = get_decoded_lines(encoded_message, key)
```

The next and final step is to open a file in write `'w'` mode, and `print()` our decoded lines into it. You can do this either by writing them, one-by-one, with a `for` -loop:

```
for line in decoded_lines:
    print(line, end='.\n', file=output_file)
```

Or by writing them all at once using the `join()` method:

```
decoded_message = ".\n".join(decoded_lines)
print(decoded_message, file=output_file)
```

Either way will work just fine.

Here's the full solution.