Assignment 2: Applications of modular arithmetics

Question 1: Cryptography

In this question you will implement two simple cryptographic algorithms - Caesar's and Vigenère ciphers.

1. Implement Caesar's cipher: https://en.wikipedia.org/wiki/Caesar_cipher, both the encoder and the decoder. Your encoder should use a right shift of 11. Your implementation should deal with both uppercase and lowercase characters. For example - `A` should be encoded as `L`, and `a` should be encoded as `l`. Non-alphabetic characters should stay as they are.
2. Demonstrate the cipher by encoding and then decoding the Zen of Python using Caesar cipher:
   - Encode the Zen of Python and print the result.
   - Decode the result of encoding and print the decoded string.
   - Compare the original and recovered texts - they should be the same.
3. For what *n*, where *n* is the shift size, both the encoder and the decoder would output the same resulting string for every given input string? In other words, find an *n* such that for each *x* we would get: $(x + n) \bmod 26 = (x - n) \bmod 26$.
4. Implement the Vigenère cipher: https://en.wikipedia.org/wiki/Vigenère_cipher, both the encoder and the decoder. Use the keyword `XYZZY`. As before, your implementation should preserve capitalization and keep non-alphabetic characters as they are. For example - the string `Hey, you!` should be encoded using the pairs `(H, X), (e, y), (y, z), (y, z), (o, y), (u, x)`, resulting with the cipher `Ecx, xmr!`.
5. Demonstrate the Vigenère cipher by encoding and then decoding the Zen of Python, as before.

```
import this  # This line actually prints out the Zen of Python. Curious? See: https://github.com/python/cpython/blob/main/Lib/this.py

# For convenience, use the following constant.
ZEN = '''The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
'''
```

```
# Solution 1
```

```
# Solution 2
```

```
# Solution 3
```

```
# Solution 4
```

```
# Solution 5
```

Question 2: Hash tables and hash functions

The code cell below contains a list of Israeli localities and a list of lists of ZIP codes, each containing one or more ZIP codes that belong to the locality placed at the same index in the list of localities (e.g., the list of ZIP codes at `zipcodes[4]` contains ZIP codes that belong to the locality at `localities[4]`).

In the first two tasks you will create two hash tables that map each ZIP code to its corresponding locality. For example,

```
locality = zipcodes_to_localities['77771']  # Using Python's dictionary
print(locality)

locality = tget(t, '24990')  # Using our implementation of dictionary
print(locality)
```

should print

```
Ashdod
Beit Jann
```

The rest of the tasks deal with hash functions.

1. Create the hash table using Python's dictionary. Print it using the function `pprint`.
2. Create the hash table using our implementation (see lecture notes) of dictionary using the built-in hash function. Print it using the function `pprint`.
3. Implement a hash function for ZIP codes, that maps each ZIP code (as a string) to the number: (m + 1) * (k + 1), where m is the maximal digit of the ZIP code, and k is the index of its first occurrence. For example, `'10727'` (m = 7, k = 2) and `'21053'` (m = 5, k = 3) should both be mapped to the number 24.
4. May the hash function you implemented in task 3 be used as a reliable checksum function? Explain your answer with an example.
5. MD5 is a widely used hash function, primarily used as a checksum. The function `md5(s)` in the code cell below takes a string and returns its MD5 digest represented as a string of 32 hexadecimal digits. You are required to write a program that finds and prints the string of 3 lowercase English alphabet characters whose MD5 digest is `0b08bd98d279b88859b628cd8c061ae0`.

```
# For tasks 1 and 2

from pprint import pprint

localities = \
[
 'Qiryat Shemona',
 'Beit Jann',
 'Harish',
 'Tira',
 'Bene Beraq',
 'Ashdod',
 'Sederot',
 'Beersheba',
 'Kseife',
 'Tzofar'
]

zipcodes = \
[
 ['11032', '11561'],
 ['24990'],
 ['37611'],
 ['44915'],
 ['51461', '51529', '51562'],
 ['77756', '77771'],
 ['87112'],
 ['84138', '84277', '84540', '84885'],
 ['84923'],
 ['86830']
]

# For task 5

import hashlib

def md5(s):
  return hashlib.md5(s.encode('utf-8')).hexdigest()
```

```
# Solution 1
```

```
# Solution 2
```

```
# Solution 3
```

```
# Solution 4
```

```
# Solution 5
```