```
# TODO 1
alphabet[7]
```

```
def isLetter(character: str):
  """Checks if the character is a letter in the alphabet

  Args:
      character (str): character to verify

  Returns:
      bool: if character is a letter
  """
  # TODO 2
  return character in alphabet

def idxOfLetter(letter: str):
  """ Function returns the index of the letter within the alphabet

  Args:
      letter (str): letter to search

  Returns:
      int: index of letter in the alphabet
  """
  # TODO 3
  return alphabet.index(letter)

def incrementIndex(index: int, k: int):
  """increments the index according to the k value, the output value stays within the range [0-25]

  Args:
      index (int): index of letter
      k (int): index shift

  Returns:
      int: new index of letter
  """
  # TODO 4
  idx = index + k
  if idx > 25:
    idx -= 26
  elif idx < 0:
    idx += 26
  return idx

def cesarEncoding(text: str, k: int):
  """Takes a text and encodes it.

  Args:
      text (str): text to be encoded
      k (int): character shift (positive and negative)

  Returns:
      str: encoded text
  """
  text_encoded = ""

  # Loop though every character in the input text
  for char in text.upper():
    # TODO 5
    # 1. Kontrollieren ob der Character ein Buchstabe ist
    if isLetter(char):
      # 2. Suchen des Index des Buchstaben
      index = idxOfLetter(char)
      # 3. Inkrementieren oder dekrementieren des Indexes um k
      new_index = incrementIndex(index, k)
      # 4. Suchen des neuen Buchstaben
      text_encoded += alphabet[new_index]
    else:
      # Append character if not a letter
      text_encoded += char

  return text_encoded


# Test of the function
plain_text = "Mit ihren Bachelor-Studiengaengen stellt die HES-SO Valais-Wallis in Sitten ein echtes Kompetenz und Innovationszentrum dar"
encoded_text = cesarEncoding(text=plain_text, k=3)
decoded_text = cesarEncoding(text=encoded_text, k=-3)

print(plain_text)
print()
print(encoded_text)
print()
print(decoded_text)
```

```
Click to hideclass Scrambler:
  def __init__(self, type_key: str = None, startpos: int = 0, custom: str = None):
    """Create a alphabet shuffle, this represents one rotor or a reflector. The configuration can be choosen from the first ever Enigma, the latest WWII Enigma machine, a Random pattern or a custom setting.
    The reflector and rotor settings are according to Wikipedia https://en.wikipedia.org/wiki/Enigma_rotor_details:

    Args:
        type_key (str): type of enigma rotor ["etw", "i", "ii", "iii", ..., "custom"]
        startpos (int, optional): rotor startposition. Defaults to 0.
        custom (string, optional): string of chars representing the custom configuration. Defaults to None.
    """
    self.alphabet = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']

    # Possible rotor configurations
    self.configs = {
      "etw"    : "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
      "i"      : "EKMFLGDQVZNTOWYHXUSPAIBRCJ",
      "ii"     : "AJDKSIRUXBLHWTMCQGZNPYFVOE",
      "iii"    : "BDFHJLCPRTXVZNYEIWGAKMUSQO",
      "iv"     : "ESOVPZJAYQUIRHXLNFTGKDCMWB",
      "v"      : "VZBRGITYUPSDNHLXAWMJQOFECK",
      "vi"     : "JPGVOUMFYQBENHZRDKASXLICTW",
      "vii"    : "NZJHGRCXMYSWBOUFAIVLPEKQDT",
      "viii"   : "FKQHTLXOCBJSPDZRAMEWNIUYGV",
      "a"      : "EJMZALYXVBWFCRQUONTSPIKHGD",
      "b"      : "YRUHQSLDPXNGOKMIEBFZCWVJAT",
      "c"      : "FVPJIAOYEDRZXWGCTKUQSBNMHL",
    }

    # TODO 6
```

```python
    self.type_key = type_key.lower().replace(" ", "").replace("_", "").replace("-", "")
    self.startpos = startpos

    # get the key
    if self.type_key == "custom":
      self.transformation = self.getConfig(custom)
    else:
      self.transformation = self.getConfig(self.configs[self.type_key])

    # setup initial position of rotors
    self.transformation = self.rol(self.transformation, self.startpos)

    self.key = self.getKey()

  def getConfig(self, str_config: str):
    """Transforms the string configuration into an int array

    Args:
        str_config (str): string of configuration. All alphabet characters need to be represented.

    Returns:
        list: list of int representing the alphabet positions of the config
    """
    config = []
    # TODO 7
    for char in str_config:
      config.append(self.alphabet.index(char))
    return config

  def getKey(self):
    """Get the key of the current transformation config

    Returns:
        str: string of characters of the current config
    """
    alphabet = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
    key = ""
    for idx in self.transformation:
      key += self.alphabet[idx]
    return key

  def rol(self,  string: str, n: int):
    """Rotating shift left of a string by n characters
       example: n=2
       "Test" => "stTe"
    Args:
        string (str): input string
        n (int): number of bits to shift

    Returns:
        str: string rotated shift left by n chars
    """
    # TODO 8
    return string[n:] + string[:n]

  def passthrough(self, idx: int):
    """Pass element through (index => element)

    Args:
        idx (int): index of character index to return

    Returns:
        int: new character index
    """
    return self.transformation[idx]

  def passthroughRev(self, elem):
    """Reverse Passthrough, enter character index and return list index

    Returns:
       int: index of character index
    """
    return self.transformation.index(elem)

  def rotate(self):
    """Rotate the rotors by one position
    """
    self.transformation = self.rol(self.transformation, 1)

  def setTransformation(self, transformation: list):
    """Set manually the tranformation. E.g. to reset the machine

    Args:
```

```python
        transformation (list): transformation list to be used
    """
    self.transformation = transformation
```

```python
class EnigmaMachine:
  def __init__(self, nb_rotors: int = 3, rotor_types: list = ["i", "iii", "iii"], rotor_startpos: list = [1, 2, 3], rotor_custom_configs: list = None, reflector_type: str = "a", plugboard_config: list = None, print_specialchars: bool = False):
    """Enigma Virtual Machine
        nb_rotors (int, optional): number of rotors in the machine. Defaults to 3.
        rotor_types (list, optional): list of types rotors types ["etw"|"i"|"ii"|"iii"|"iv"|"v"|"vi"|"vii"|"viii"]. Needs to be size of nb_rotors. Defaults to ["i", "ii", "iii"].
        rotor_startpos (list, optional): list of int representing thestart positions of the rotors. Needs to be the size of nb_rotors. Defaults to [1, 2, 3].
        rotor_custom_configs (list, optional): list of int lists representing the custom rotor configuration, only needed if "custom" type is choosen. Needs to be the size of nb_rotors if used. Defaults to None
        reflector_type (str, optional): type of reflector ["a"|"b"|"c"]. Defaults to "a".
        plugboard_config (list, optional): list of character combinations. Defaults to None, will result in A<->Z, B<->Y, ...
        print_specialchars (bool, optional): Print characters missing by enigma. Defaults to False.
    """
    self.nb_rotors = nb_rotors
    self.rotor_types = rotor_types
    self.rotor_startpos = rotor_startpos
    self.rotor_custom_configs = rotor_custom_configs
    self.reflector_type = reflector_type
    self.printspecialchars = print_specialchars
    if plugboard_config is None:
      self.plugboard_config = ["AZ", "BY", "CX", "DW", "EV", "FU", "GT", "HS", "IR", "JQ", "KP", "LO", "MN"]
    else:
      self.plugboard_config = plugboard_config

    # create the rotors and reflector
    self.rotors = []
    self.original_rotors = []
    self.reflector = None
    self.plugboard = None

    self.setupRotors()
    self.setupReflector()
    self.setupPlugboard()

  def setupRotors(self):
    """Setup the rotors configuration
    """
    for i in range(self.nb_rotors):
      if self.rotor_custom_configs is None:
        self.rotors.append(Scrambler(self.rotor_types[i], self.rotor_startpos[i]))
      else:
        self.rotors.append(Scrambler(self.rotor_types[i], self.rotor_startpos[i], self.rotor_custom_configs[i]))
      self.original_rotors.append(self.rotors[i].transformation)

  def setupReflector(self):
    """Setup the reflector
    """
    self.reflector = Scrambler(self.reflector_type)

  def setupPlugboard(self):
    """Setup the plugboard"""
    # Transform into scrambler key
    alphabet = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
    key = " " * 26
    for elem in self.plugboard_config:
      # TODO 9
      key = key[:alphabet.index(elem[0])] + elem[1] + key[alphabet.index(elem[0])+1:]
      key = key[:alphabet.index(elem[1])] + elem[0] + key[alphabet.index(elem[1])+1:]

    self.plugboard = Scrambler("custom", 0, key)

  def printEnigmaSetup(self):
    """Print Enigma setup of plugboard, rotors and reflector
    """
    print("Enigma Setup")
    print("============\n")

    # TODO 10
    for i in range(self.nb_rotors):
      print("* Rotor {}".format(i))
      print("  - Type      : {}".format(self.rotors[i].type_key))
      print("  - Key       : {}".format(self.rotors[i].key))
      print("  - StartPos  : {}".format(self.rotors[i].startpos))

    print("* Reflector")
    print("  - Type      : {}".format(self.reflector.type_key))
    if self.reflector.type_key == "custom" or self.reflector.type_key == "random":
      print("  - Key       : {}".format(self.reflector.key))
      print("  - StartPos  : {}".format(self.reflector.startpos))

    print("* Plugboard")
    print("  - Key       : {}".format(self.plugboard_config))

  def reset(self):
    """Restart the original rotor start positions
    """
    for i in range(0, self.nb_rotors):
      self.rotors[i].setTransformation(self.original_rotors[i])

  def encode(self, text: str):
    """Encode and decode a string

    Args:
        text (str): string to encode

    Returns:
        str : depending on the input string, the encoded or decoded output
    """
    ln = 0
    encrypted_text = ""
    for l in text.lower():
      # get char position in alphabet
      num = ord(l) % 97
      if (num > 25 or num < 0):
        # Special character
        if (self.printspecialchars):
          encrypted_text += l
      else:
        # encodable character
        ln += 1

        # TODO 11
        # pass through plugboard
        num = self.plugboard.passthrough(num)

        # pass through rotors
        for i in range(0, self.nb_rotors):
          num = self.rotors[i].passthrough(num)

        # reflected by the reflector
        num = self.reflector.passthrough(num)

        # pass through rotors from the other side
        for i in range(0, self.nb_rotors):
          num = self.rotors[self.nb_rotors - i - 1].passthroughRev(num)

        # pass through plugboard from the other side
        num = self.plugboard.passthroughRev(num)

        # Encode character
        encrypted_text += "" + chr(97 + num)

        # rotate the rotors
        for i in range(0, self.nb_rotors):
          if (ln % ((i * 6) + 1) == 0):
            self.rotors[i].rotate()
    return encrypted_text
```