2024 4-1 Information Security Team Project

Final

20102104 Kim Sua

20102117 Lee Jeonghyeon

Contents

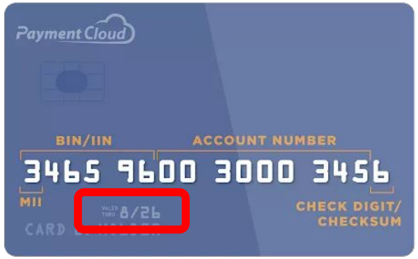
- Proposal
- Implementation
    - Card Brand Check
    - Expiry Date
    - Card Validity
- Performance
- Program Execution

Plan

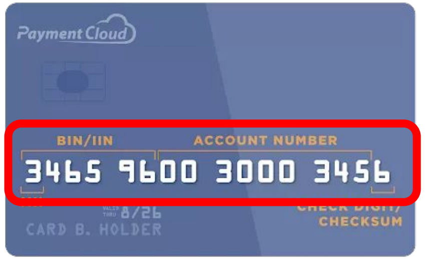
1. Verify Card Types (Brand)

Interac
VISA
AMERICAN EXPRESS
UnionPay
银联
DISCOVER
Diners Club INTERNATIONAL
JCB
Pay
G Pay

2. Check the validity period



3. Verify card number



Step 1 Verify Card Types

- The first digit of the card number determines the card types
  - Starting with 4 : VISA
  - Starting with 5 : Mastercard
  - Starting with 9 : Domestic

- Using comparative operations of homomorphic encryption to distinguish whether the inputted card is a Mastercard, Visa, or Domestic card.

Step 2 Verify Card Valid Period

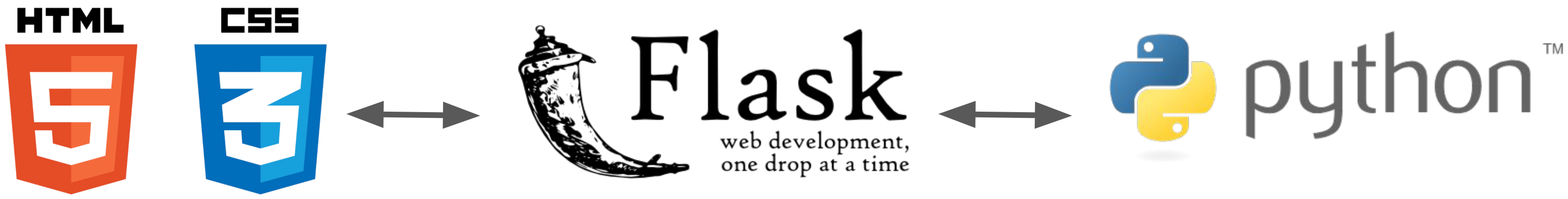
- Verification of the validity period inputted on the card by comparing it with the current date to ensure its validity.

Step 3 Verify Card Number Validity

- Verify the Card number validity using the algorithm
- Multiply each digit in the odd positions of the card number by 2. If the resulting number is a two-digit number, add the digits together.
- Sum up all the digits in the even positions.
- Add the sums from steps 1 and 2 together.
- Add a specific number to the result from step 3 and adjust the 16th digit of the card number to ensure it is divisible by 10.

Implementation - Overview

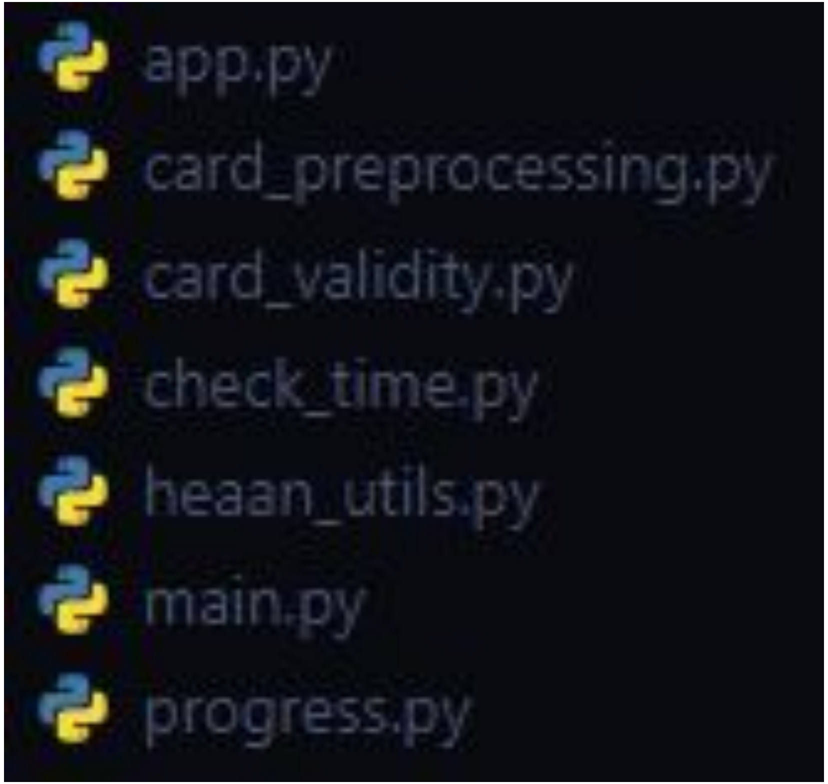
This web application architecture leverages the strengths of HTML for building user interfaces, Flask for handling HTTP requests and server-side routing, and Python for implementing business logic and processing data.



Implementation - Detailed Design

Application Structure

- **app.py**:
  The main entry point of the Flask application. Handles routing and starts the server.
- **main.py**:
  Handles the input from the HTML form and orchestrates the processing workflow.
- **card_preprocessing.py**:
  Contains functions to preprocess the card data.
- **card_validity.py**:
  Contains functions to validate the card data.
- **heaan_utils.py**:
  Utilizes the Pi-heaan library for secure computations.

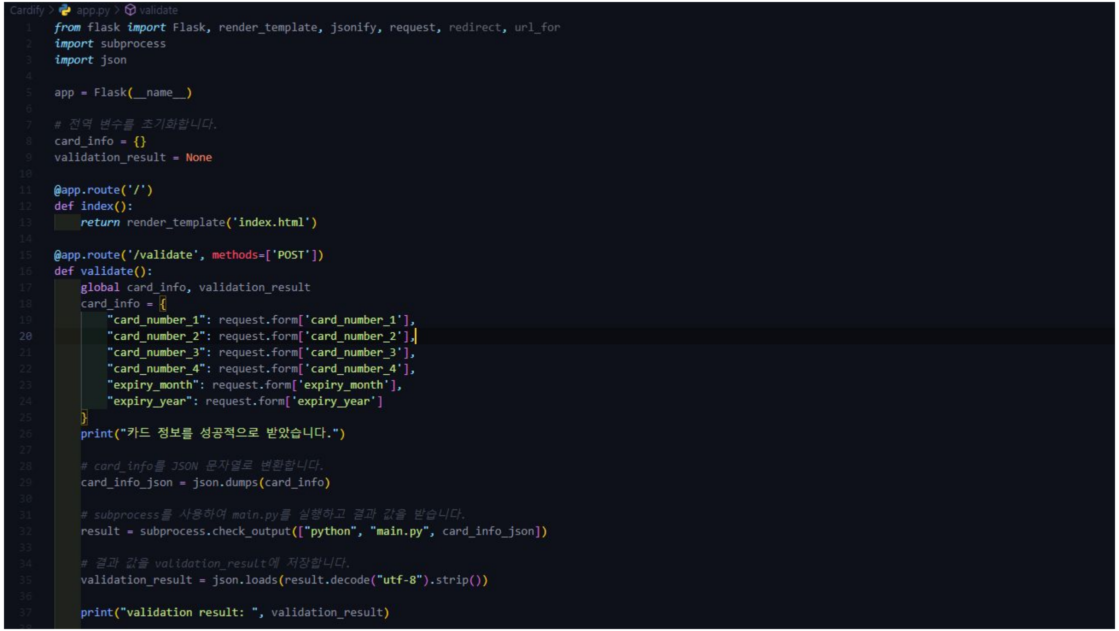


app.py
card_preprocessing.py
card_validity.py
check_time.py
heaan_utils.py
main.py
progress.py

Implementation - Detailed Design

app.py

Set up the Flask application, define routes, and render the initial HTML page.



```python
from flask import Flask, render_template, jsonify, request, redirect, url_for
import subprocess
import json

app = Flask(__name__)

# 전역 변수를 초기화합니다.
card_info = {}
validation_result = None

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/validate', methods=['POST'])
def validate():
    global card_info, validation_result
    card_info = {
        "card_number_1": request.form['card_number_1'],
        "card_number_2": request.form['card_number_2'],
        "card_number_3": request.form['card_number_3'],
        "card_number_4": request.form['card_number_4'],
        "expiry_month": request.form['expiry_month'],
        "expiry_year": request.form['expiry_year']
    }
    print("카드 정보를 성공적으로 받았습니다.")

    # card_info를 JSON 문자열로 변환합니다.
    card_info_json = json.dumps(card_info)

    # subprocess를 사용하여 main.py를 실행하고 결과 값을 받습니다.
    result = subprocess.check_output(["python", "main.py", card_info_json])

    # 결과 값을 validation_result에 저장합니다.
    validation_result = json.loads(result.decode("utf-8").strip())

    print("validation result: ", validation_result)
```

Implementation - Detailed Design

app.py

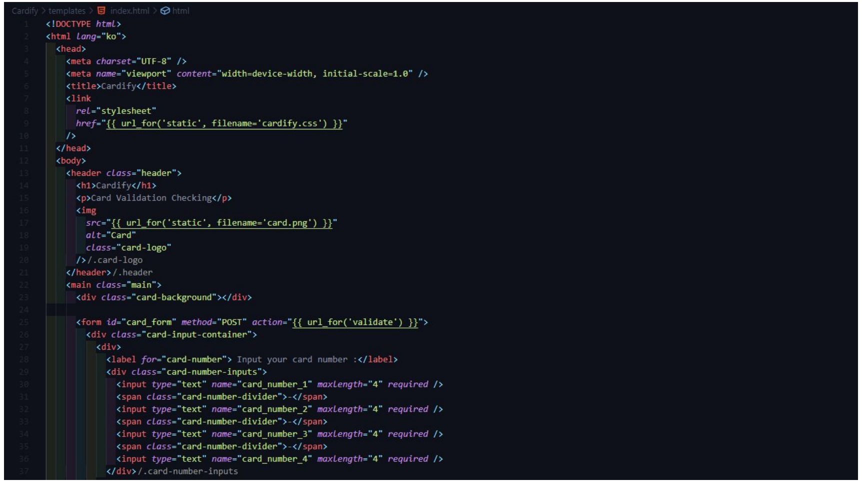
- Form Handling:
  The /validate route collects card information from the submitted form, processes it using a separate script (main.py), and updates the global variable validation_result with the processed data.
- Subprocess Usage:
  The subprocess.check_output function is used to run main.py with the card information passed as a JSON string. The output from main.py is captured and decoded into a JSON object.
- Template Rendering:
  The render_template function is used to render HTML templates (index.html and result.html). The result.html template is dynamically updated with the validation result.
- Global Variables:
  The use of global variables (card_info and validation_result) allows the application to store and access data across different routes.

Implementation - Detailed Design

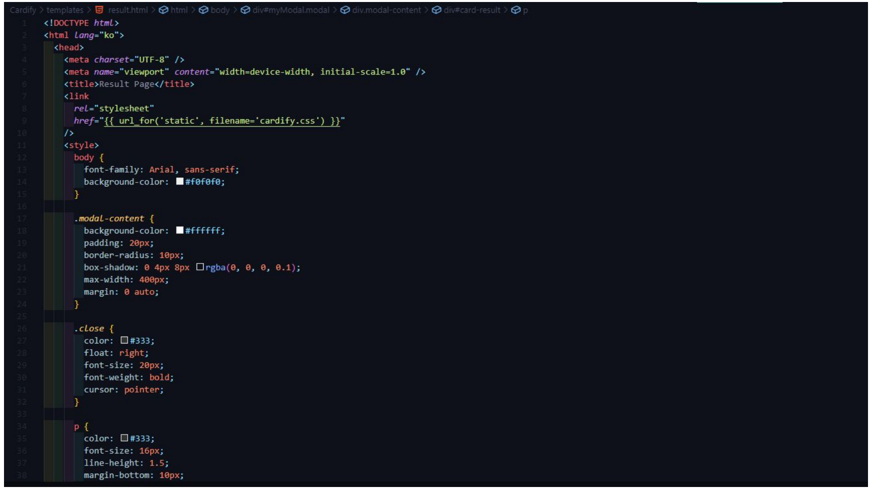
templates/index.html & result.html

Provide the user interface

- Index.html : input card information
- result.html : show the results



```html
<!DOCTYPE html>
<html lang="ko">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Cardify</title>
    <link
      rel="stylesheet"
      href="{{ url_for('static', filename='cardify.css') }}"
    />
  </head>
  <body>
    <header class="header">
      <h1>Cardify</h1>
      <p>Card Validation Checking</p>
      <img
        src="{{ url_for('static', filename='card.png') }}"
        alt="Card"
        class="card-logo"
      />/.card-logo
    </header>/.header
    <main class="main">
      <div class="card-background"></div>

      <form id="card_form" method="POST" action="{{ url_for('validate') }}">
        <div class="card-input-container">
          <div>
            <label for="card-number"> Input your card number :</label>
            <div class="card-number-inputs">
              <input type="text" name="card_number_1" maxlength="4" required />
              <span class="card-number-divider">-</span>
              <input type="text" name="card_number_2" maxlength="4" required />
              <span class="card-number-divider">-</span>
              <input type="text" name="card_number_3" maxlength="4" required />
              <span class="card-number-divider">-</span>
              <input type="text" name="card_number_4" maxlength="4" required />
            </div>/.card-number-inputs
```



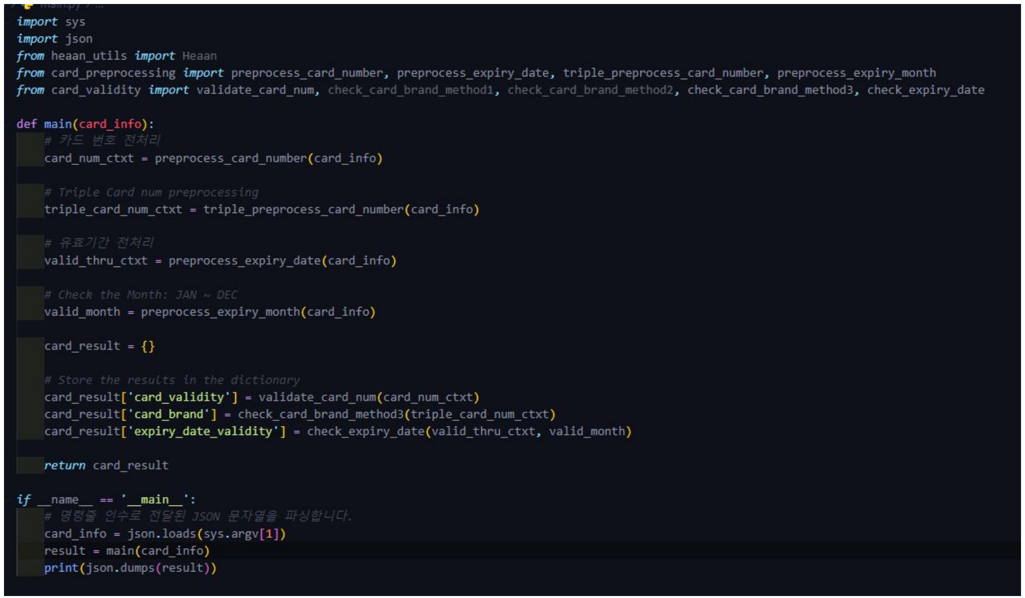
Cardify > templates > result.html > html > body > div#myModal.modal > div.modal-content > div#card-result > p

```html
<!DOCTYPE html>
<html lang="ko">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Result Page</title>
    <link
      rel="stylesheet"
      href="{{ url_for('static', filename='cardify.css') }}"
    />
    <style>
      body {
        font-family: Arial, sans-serif;
        background-color: #f0f0f0;
      }

      .modal-content {
        background-color: #ffffff;
        padding: 20px;
        border-radius: 10px;
        box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);
        max-width: 400px;
        margin: 0 auto;
      }

      .close {
        color: #333;
        float: right;
        font-size: 20px;
        font-weight: bold;
        cursor: pointer;
      }

      p {
        color: #333;
        font-size: 16px;
        line-height: 1.5;
        margin-bottom: 10px;
```

Implementation - Detailed Design

main.py

Receive input from the form, invoke preprocessing and validation functions, and return the result.



```python
import sys
import json
from heaan_utils import Heaan
from card_preprocessing import preprocess_card_number, preprocess_expiry_date, triple_preprocess_card_number, preprocess_expiry_month
from card_validity import validate_card_num, check_card_brand_method1, check_card_brand_method2, check_card_brand_method3, check_expiry_date

def main(card_info):
    # 카드 번호 전처리
    card_num_ctxt = preprocess_card_number(card_info)

    # Triple Card num preprocessing
    triple_card_num_ctxt = triple_preprocess_card_number(card_info)

    # 유효기간 전처리
    valid_thru_ctxt = preprocess_expiry_date(card_info)

    # Check the Month: JAN ~ DEC
    valid_month = preprocess_expiry_month(card_info)

    card_result = {}

    # Store the results in the dictionary
    card_result['card_validity'] = validate_card_num(card_num_ctxt)
    card_result['card_brand'] = check_card_brand_method3(triple_card_num_ctxt)
    card_result['expiry_date_validity'] = check_expiry_date(valid_thru_ctxt, valid_month)

    return card_result

if __name__ == '__main__':
    # 명령줄 인수로 전달된 JSON 문자열을 파싱합니다.
    card_info = json.loads(sys.argv[1])
    result = main(card_info)
    print(json.dumps(result))
```

Implementation - Detailed Design

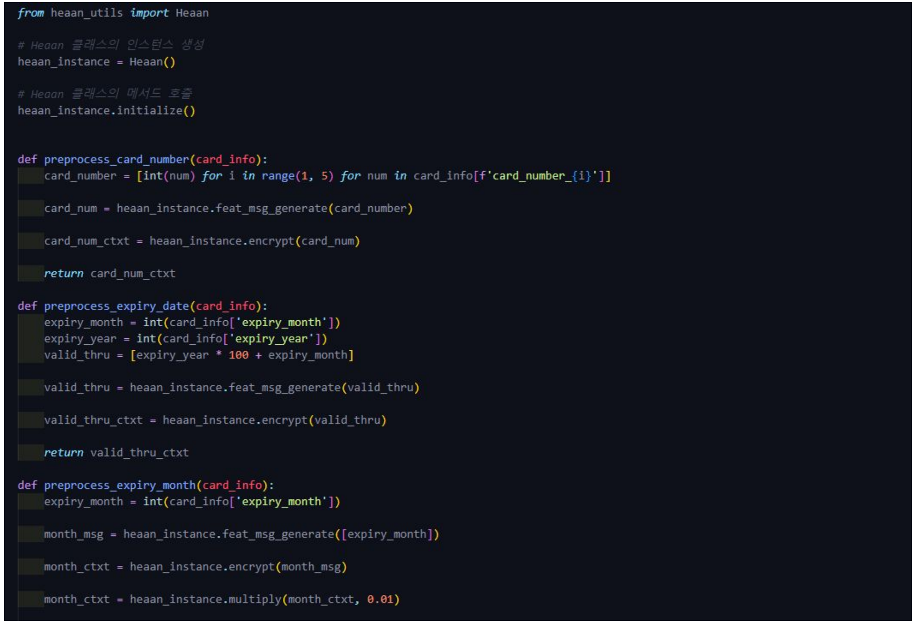
main.py

- Functionality:
  The main function orchestrates the entire workflow: preprocessing card information, validating it, and aggregating results into a dictionary.
- Preprocessing:
  - The card number is preprocessed and encrypted using preprocess_card_number.
  - The card number is tripled, preprocessed, and encrypted using triple_preprocess_card_number.
  - The expiration date is preprocessed and encrypted using preprocess_expiry_date.
  - The expiration month is preprocessed, encrypted, and scaled using preprocess_expiry_month.
- Validation:
  - The card number's validity is checked using validate_card_num.
  - The card brand is checked using check_card_brand_method3.
  - The expiration date validity is checked using check_expiry_date.
- Command-Line Interface:
  - The script is designed to be executed from the command line, with card information passed as a JSON string argument.
  - The script outputs the validation results in JSON format, suitable for further processing or display.

Implementation - Detailed Design

card_preprocessing.py

Preprocess the card number and expiration date.



```python
from heaan_utils import Heaan

# Heaan 클래스의 인스턴스 생성
heaan_instance = Heaan()

# Heaan 클래스의 메서드 호출
heaan_instance.initialize()


def preprocess_card_number(card_info):
    card_number = [int(num) for i in range(1, 5) for num in card_info[f'card_number_{i}']]

    card_num = heaan_instance.feat_msg_generate(card_number)

    card_num_ctxt = heaan_instance.encrypt(card_num)

    return card_num_ctxt

def preprocess_expiry_date(card_info):
    expiry_month = int(card_info['expiry_month'])
    expiry_year = int(card_info['expiry_year'])
    valid_thru = [expiry_year * 100 + expiry_month]

    valid_thru = heaan_instance.feat_msg_generate(valid_thru)

    valid_thru_ctxt = heaan_instance.encrypt(valid_thru)

    return valid_thru_ctxt

def preprocess_expiry_month(card_info):
    expiry_month = int(card_info['expiry_month'])

    month_msg = heaan_instance.feat_msg_generate([expiry_month])

    month_ctxt = heaan_instance.encrypt(month_msg)

    month_ctxt = heaan_instance.multiply(month_ctxt, 0.01)
```

Implementation - Detailed Design

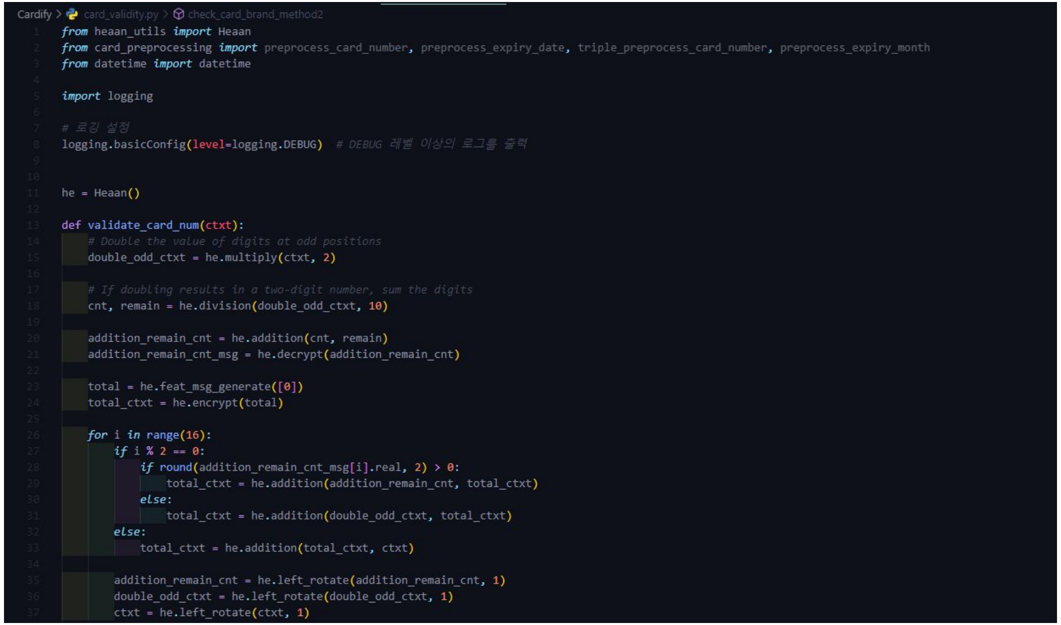
card_preprocessing.py

- **Initialization:**
  An instance of the Heaan class is created and initialized to prepare the HEAAN library for processing and encryption tasks.
- **Function Descriptions:**
  Each function processes specific parts of the card information and encrypts them using HEAAN's methods. These functions are designed to handle the card number and expiration date securely.
- **Feature Message Generation and Encryption**:
  - The feat_msg_generate method of the Heaan instance is used to convert data into a format suitable for encryption.
  - The encrypt method encrypts the processed data.
  - The multiply method scales the encrypted data in the preprocess_expiry_month function.

Implementation - Detailed Design

card_validity.py

Validate the preprocessed card data.



```python
from heaan_utils import Heaan
from card_preprocessing import preprocess_card_number, preprocess_expiry_date, triple_preprocess_card_number, preprocess_expiry_month
from datetime import datetime

import logging

# 로깅 설정
logging.basicConfig(level=logging.DEBUG)  # DEBUG 레벨 이상의 로그를 출력


he = Heaan()

def validate_card_num(ctxt):
    # Double the value of digits at odd positions
    double_odd_ctxt = he.multiply(ctxt, 2)

    # If doubling results in a two-digit number, sum the digits
    cnt, remain = he.division(double_odd_ctxt, 10)

    addition_remain_cnt = he.addition(cnt, remain)
    addition_remain_cnt_msg = he.decrypt(addition_remain_cnt)

    total = he.feat_msg_generate([0])
    total_ctxt = he.encrypt(total)

    for i in range(16):
        if i % 2 == 0:
            if round(addition_remain_cnt_msg[i].real, 2) > 0:
                total_ctxt = he.addition(addition_remain_cnt, total_ctxt)
            else:
                total_ctxt = he.addition(double_odd_ctxt, total_ctxt)
        else:
            total_ctxt = he.addition(total_ctxt, ctxt)

        addition_remain_cnt = he.left_rotate(addition_remain_cnt, 1)
        double_odd_ctxt = he.left_rotate(double_odd_ctxt, 1)
        ctxt = he.left_rotate(ctxt, 1)
```

Implementation - Detailed Design

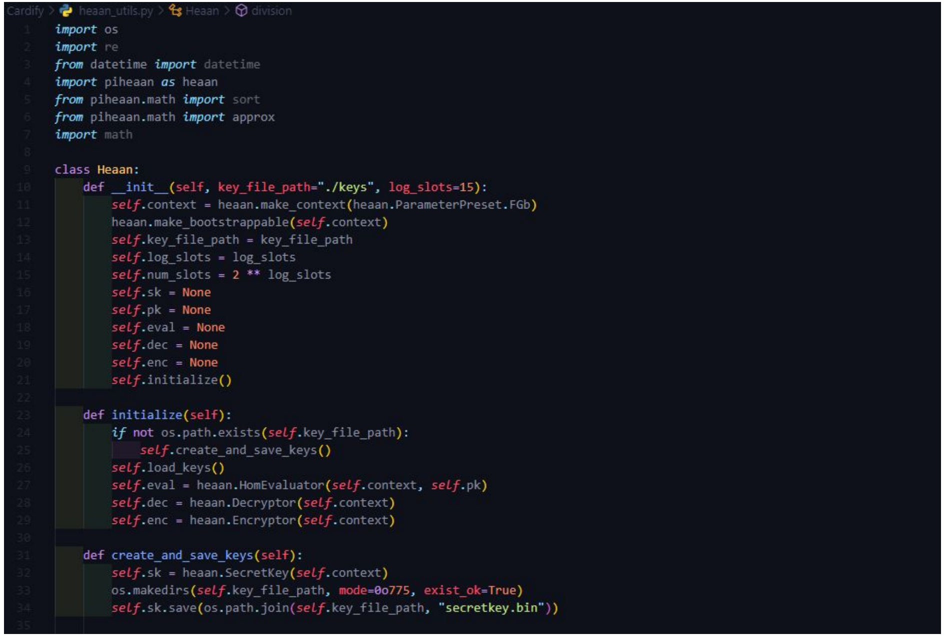
card_validity.py

- validate_card_num(ctxt):
    - This function validates a card number using the Luhn algorithm.
    - It doubles the value of digits at odd positions and sums the digits if doubling results in a two-digit number.
    - The total sum of all digits is then checked to see if it is divisible by 10 to determine the validity of the card number.
- check_card_brand_method1(ctxt):
    - This function checks the brand of a card by comparing the first digit of the card number to known prefixes (4 for Visa, 5 for MasterCard, 9 for Domestic cards).
- check_card_brand_method2(ctxt):
    - This function checks the brand of a card using a different method by rotating the prefix and comparing it to the card number.
- check_card_brand_method3(ctxt):
    - This function checks the brand of a card by subtracting the prefix from the card number and checking for equality to zero.
- check_expiry_date(ctxt, month_ctxt):
    - This function checks if the card's expiry date is valid by comparing it to the current date and ensuring the month is within the range of 1 to 12.

Implementation - Detailed Design

heaan_utils.py

Utilize PI-HEAAN library functions for secure computations.



```python
import os
import re
from datetime import datetime
import piheaan as heaan
from piheaan.math import sort
from piheaan.math import approx
import math

class Heaan:
    def __init__(self, key_file_path="./keys", log_slots=15):
        self.context = heaan.make_context(heaan.ParameterPreset.FGb)
        heaan.make_bootstrappable(self.context)
        self.key_file_path = key_file_path
        self.log_slots = log_slots
        self.num_slots = 2 ** log_slots
        self.sk = None
        self.pk = None
        self.eval = None
        self.dec = None
        self.enc = None
        self.initialize()

    def initialize(self):
        if not os.path.exists(self.key_file_path):
            self.create_and_save_keys()
        self.load_keys()
        self.eval = heaan.HomEvaluator(self.context, self.pk)
        self.dec = heaan.Decryptor(self.context)
        self.enc = heaan.Encryptor(self.context)

    def create_and_save_keys(self):
        self.sk = heaan.SecretKey(self.context)
        os.makedirs(self.key_file_path, mode=0o775, exist_ok=True)
        self.sk.save(os.path.join(self.key_file_path, "secretkey.bin"))

```

Implementation - Detailed Design

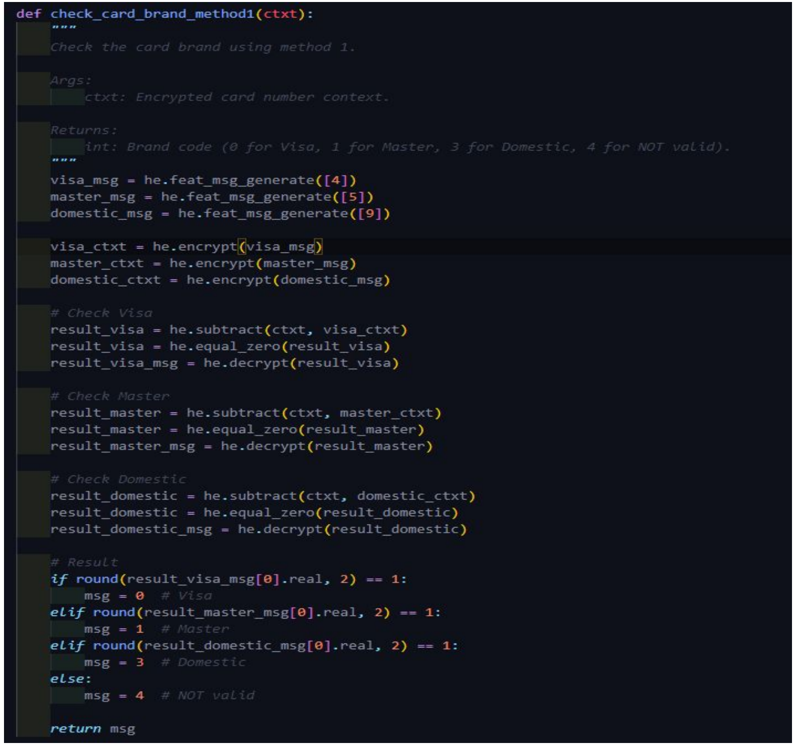
heaan_utils.py

- Homomorphic Operations:
  - check_card_brand(self, card_num_ctxt): Checks the brand of a credit card based on the encrypted card number.
  - feat_msg_generate(self, feat): Generates a message from a feature array.
  - encrypt(self, plaintext): Encrypts a plaintext list.
  - decrypt(self, ciphertext): Decrypts a ciphertext.
  - division(self, divided, divider): Divides one encrypted number by another.
  - multiply(self, ctxt, factor): Multiplies an encrypted number by a constant factor.
  - subtract(self, ctxt1, ctxt2): Subtracts one encrypted number from another.
  - addition(self, ctxt1, ctxt2): Adds two encrypted numbers.
  - equal_zero(self, ctxt): Checks if an encrypted number is zero.
  - left_rotate(self, ctxt, rotation_amount): Rotates an encrypted number to the left.
  - right_rotate(self, ctxt, rotation_amount): Rotates an encrypted number to the right.
  - comparing(self, ctxt1, ctxt2): Compares two encrypted numbers.

Implementation - Card Types

Method 1

- Encrypt predefined messages representing Visa, MasterCard, and Domestic brands.
- Subtract each encrypted message from the provided encrypted card number context.
- Decrypt the results of the subtractions and check for matches.

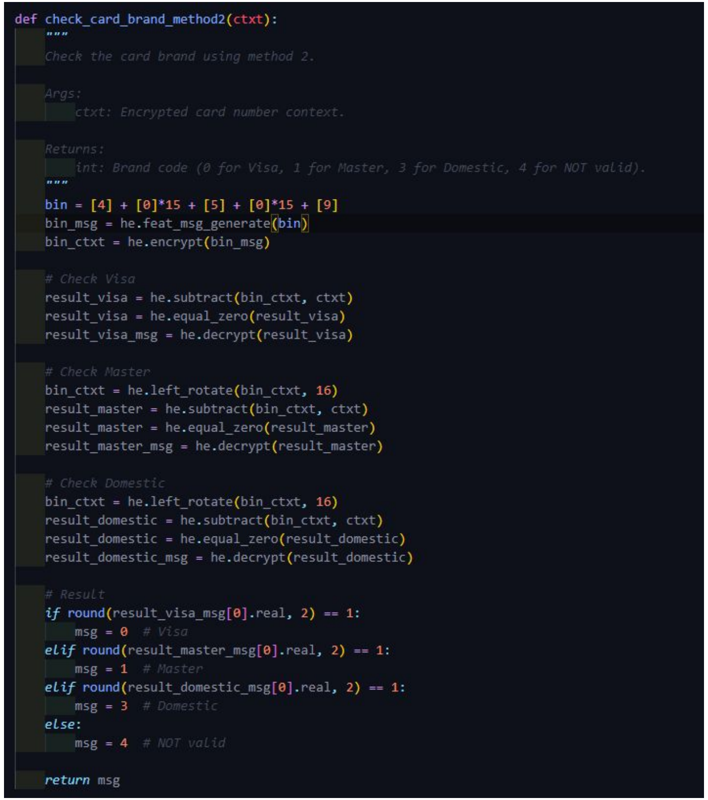


```python
def check_card_brand_method1(ctxt):
    """
    Check the card brand using method 1.

    Args:
        ctxt: Encrypted card number context.

    Returns:
        int: Brand code (0 for Visa, 1 for Master, 3 for Domestic, 4 for NOT valid).
    """
    visa_msg = he.feat_msg_generate([4])
    master_msg = he.feat_msg_generate([5])
    domestic_msg = he.feat_msg_generate([9])

    visa_ctxt = he.encrypt(visa_msg)
    master_ctxt = he.encrypt(master_msg)
    domestic_ctxt = he.encrypt(domestic_msg)

    # Check Visa
    result_visa = he.subtract(ctxt, visa_ctxt)
    result_visa = he.equal_zero(result_visa)
    result_visa_msg = he.decrypt(result_visa)

    # Check Master
    result_master = he.subtract(ctxt, master_ctxt)
    result_master = he.equal_zero(result_master)
    result_master_msg = he.decrypt(result_master)

    # Check Domestic
    result_domestic = he.subtract(ctxt, domestic_ctxt)
    result_domestic = he.equal_zero(result_domestic)
    result_domestic_msg = he.decrypt(result_domestic)

    # Result
    if round(result_visa_msg[0].real, 2) == 1:
        msg = 0  # Visa
    elif round(result_master_msg[0].real, 2) == 1:
        msg = 1  # Master
    elif round(result_domestic_msg[0].real, 2) == 1:
        msg = 3  # Domestic
    else:
        msg = 4  # NOT valid

    return msg
```

Implementation - Card Types

Method 2

- Create a binary representation of known card brands (Visa, MasterCard, Domestic).
- Encrypt the binary message.
- Subtract the encrypted binary message from the provided encrypted card number context.
- Decrypt the results of the subtractions and check for matches.

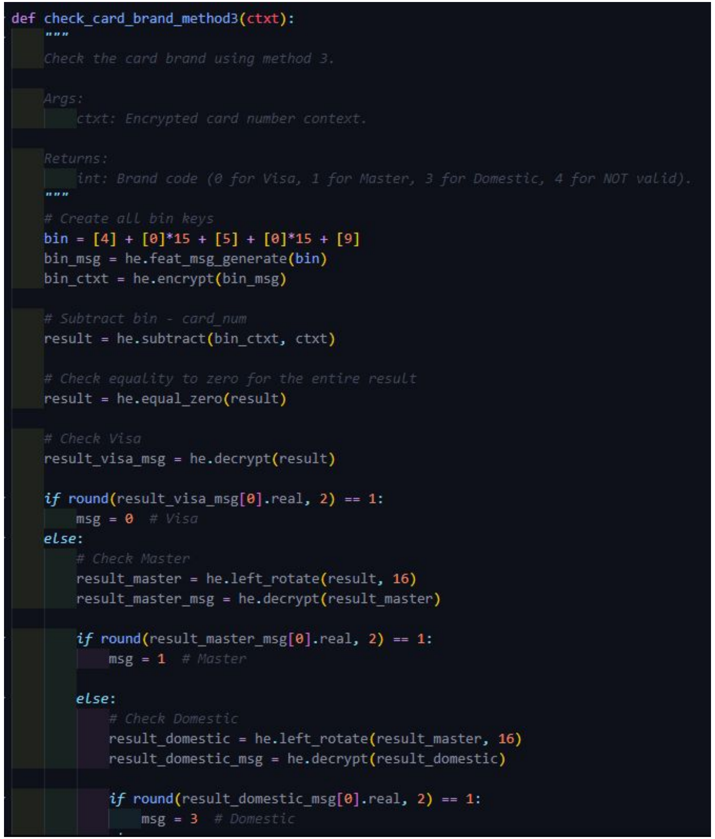


```python
def check_card_brand_method2(ctxt):
    """
    Check the card brand using method 2.

    Args:
        ctxt: Encrypted card number context.

    Returns:
        int: Brand code (0 for Visa, 1 for Master, 3 for Domestic, 4 for NOT valid).
    """
    bin = [4] + [0]*15 + [5] + [0]*15 + [9]
    bin_msg = he.feat_msg_generate(bin)
    bin_ctxt = he.encrypt(bin_msg)

    # Check Visa
    result_visa = he.subtract(bin_ctxt, ctxt)
    result_visa = he.equal_zero(result_visa)
    result_visa_msg = he.decrypt(result_visa)

    # Check Master
    bin_ctxt = he.left_rotate(bin_ctxt, 16)
    result_master = he.subtract(bin_ctxt, ctxt)
    result_master = he.equal_zero(result_master)
    result_master_msg = he.decrypt(result_master)

    # Check Domestic
    bin_ctxt = he.left_rotate(bin_ctxt, 16)
    result_domestic = he.subtract(bin_ctxt, ctxt)
    result_domestic = he.equal_zero(result_domestic)
    result_domestic_msg = he.decrypt(result_domestic)

    # Result
    if round(result_visa_msg[0].real, 2) == 1:
        msg = 0  # Visa
    elif round(result_master_msg[0].real, 2) == 1:
        msg = 1  # Master
    elif round(result_domestic_msg[0].real, 2) == 1:
        msg = 3  # Domestic
    else:
        msg = 4  # NOT valid

    return msg
```

Implementation - Card Types

Method 3

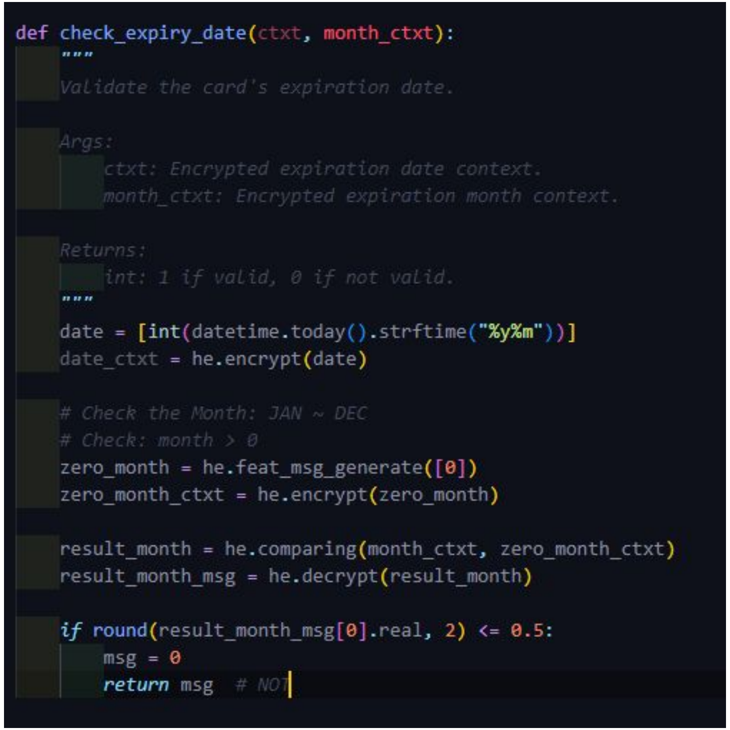
- Create a binary representation of known card brands (Visa, MasterCard, Domestic).
- Encrypt the binary message.
- Subtract the encrypted binary message from the provided encrypted card number context.
- Check if the result is equal to zero.
    - If not zero:
        - Left rotate the result by 16 bits and check for MasterCard.
    - If not MasterCard:
    - Left rotate the result by 16 bits again and check for Domestic.
    - If not Domestic, mark as NOT valid.



```python
def check_card_brand_method3(ctxt):
    """
    Check the card brand using method 3.

    Args:
        ctxt: Encrypted card number context.

    Returns:
        int: Brand code (0 for Visa, 1 for Master, 3 for Domestic, 4 for NOT valid).
    """
    # Create all bin keys
    bin = [4] + [0]*15 + [5] + [0]*15 + [9]
    bin_msg = he.feat_msg_generate(bin)
    bin_ctxt = he.encrypt(bin_msg)

    # Subtract bin - card_num
    result = he.subtract(bin_ctxt, ctxt)

    # Check equality to zero for the entire result
    result = he.equal_zero(result)

    # Check Visa
    result_visa_msg = he.decrypt(result)

    if round(result_visa_msg[0].real, 2) == 1:
        msg = 0  # Visa
    else:
        # Check Master
        result_master = he.left_rotate(result, 16)
        result_master_msg = he.decrypt(result_master)

        if round(result_master_msg[0].real, 2) == 1:
            msg = 1  # Master

        else:
            # Check Domestic
            result_domestic = he.left_rotate(result_master, 16)
            result_domestic_msg = he.decrypt(result_domestic)

            if round(result_domestic_msg[0].real, 2) == 1:
                msg = 3  # Domestic
```

Implementation - Expiry Date

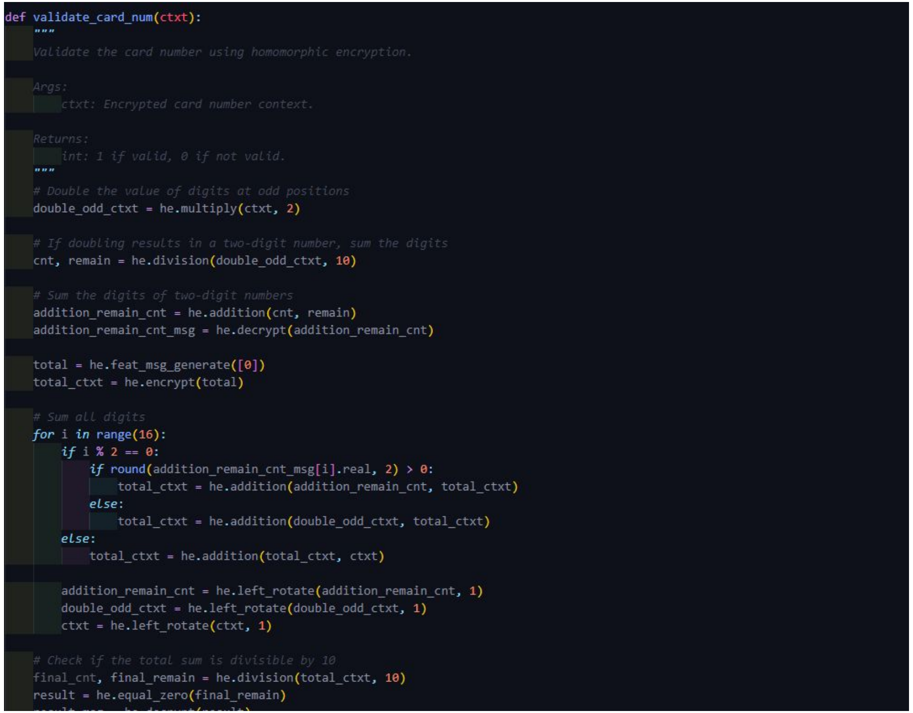
- Get the current date and format it as YYMM.
- Encrypt the current date.
- Check if the expiration month is greater than zero.
- Returns:
  - Integer code indicating validity:
    - 1 if valid (expiration month greater than zero).
    - 0 if not valid (expiration month not greater than zero).



```python
def check_expiry_date(ctxt, month_ctxt):
    """
    Validate the card's expiration date.

    Args:
        ctxt: Encrypted expiration date context.
        month_ctxt: Encrypted expiration month context.

    Returns:
        int: 1 if valid, 0 if not valid.
    """
    date = [int(datetime.today().strftime("%y%m"))]
    date_ctxt = he.encrypt(date)

    # Check the Month: JAN ~ DEC
    # Check: month > 0
    zero_month = he.feat_msg_generate([0])
    zero_month_ctxt = he.encrypt(zero_month)

    result_month = he.comparing(month_ctxt, zero_month_ctxt)
    result_month_msg = he.decrypt(result_month)

    if round(result_month_msg[0].real, 2) <= 0.5:
        msg = 0
        return msg  # NOT
```

Implementation - Card Validity

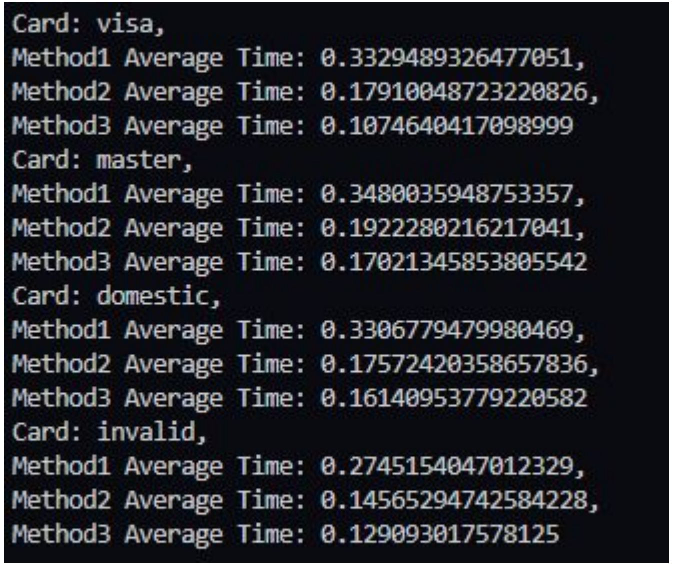
- Double the value of digits at odd positions.
- If doubling results in a two-digit number, sum the digits.
- Sum all digits of the card number.
- Check if the total sum is divisible by 10.
- Returns:
    - Integer code indicating validity:
        - 1 if valid (total sum divisible by 10).
        - 0 if not valid (total sum not divisible by 10).



```python
def validate_card_num(ctxt):
    """
    Validate the card number using homomorphic encryption.

    Args:
        ctxt: Encrypted card number context.

    Returns:
        int: 1 if valid, 0 if not valid.
    """
    # Double the value of digits at odd positions
    double_odd_ctxt = he.multiply(ctxt, 2)

    # If doubling results in a two-digit number, sum the digits
    cnt, remain = he.division(double_odd_ctxt, 10)

    # Sum the digits of two-digit numbers
    addition_remain_cnt = he.addition(cnt, remain)
    addition_remain_cnt_msg = he.decrypt(addition_remain_cnt)

    total = he.feat_msg_generate([0])
    total_ctxt = he.encrypt(total)

    # Sum all digits
    for i in range(16):
        if i % 2 == 0:
            if round(addition_remain_cnt_msg[i].real, 2) > 0:
                total_ctxt = he.addition(addition_remain_cnt, total_ctxt)
            else:
                total_ctxt = he.addition(double_odd_ctxt, total_ctxt)
        else:
            total_ctxt = he.addition(total_ctxt, ctxt)

        addition_remain_cnt = he.left_rotate(addition_remain_cnt, 1)
        double_odd_ctxt = he.left_rotate(double_odd_ctxt, 1)
        ctxt = he.left_rotate(ctxt, 1)

    # Check if the total sum is divisible by 10
    final_cnt, final_remain = he.division(total_ctxt, 10)
    result = he.equal_zero(final_remain)
    result_msg = he.decrypt(result)
```

Performance: Method 1 vs. 2 vs. 3

- **Method 3** consistently shows the lowest average processing time across all card brands compared to Method 1 and Method 2.
- Method 2 generally performs better than Method 1 but is slightly slower than Method 3.
- For all card brands, there's a noticeable improvement in efficiency when transitioning from Method 1 to Method 2, and another significant improvement when transitioning from Method 2 to Method 3.
- **Method 3 demonstrates the most efficient performance**, followed by Method 2, and lastly Method 1. This suggests that the optimization strategies implemented in Method 3 significantly enhance the processing efficiency compared to the other two methods.



```
Card: visa,
Method1 Average Time: 0.3329489326477051,
Method2 Average Time: 0.17910048723220826,
Method3 Average Time: 0.1074640417098999
Card: master,
Method1 Average Time: 0.3480035948753357,
Method2 Average Time: 0.1922280216217041,
Method3 Average Time: 0.17021345853805542
Card: domestic,
Method1 Average Time: 0.3306779479980469,
Method2 Average Time: 0.17572420358657836,
Method3 Average Time: 0.16140953779220582
Card: invalid,
Method1 Average Time: 0.2745154047012329,
Method2 Average Time: 0.14565294742584228,
Method3 Average Time: 0.129093017578125
```

Execution - Windows

1. Clone the Repository:

   ```
   git clone https://github.com/jeonghyeonee/Cardify.git
   ```

2. Setup Virtual Environment:

   ```
   python -m venv your_virtual_environment_name

   your_virtual_environment_name\Scripts\activate

   pip install -r requirements.txt
   ```

Execution - Linux/MacOS

1. Clone the Repository:

```
git clone https://github.com/jeonghyeonee/Cardify.git
```

2. Setup Virtual Environment:

```
python3 -m venv your_virtual_environment_name
source your_virtual_environment_name/bin/activate
pip install -r requirements.txt
```

Execution

1. Run the Web Application:

```
python app.py
```

2. Access the Web Interface:

Open your web browser and navigate to (http://localhost:5000)

3. Input Credit Card Information:

Follow the prompts on the web page to input the credit card number and expiry date in the specified format.

4. View Validation Result:

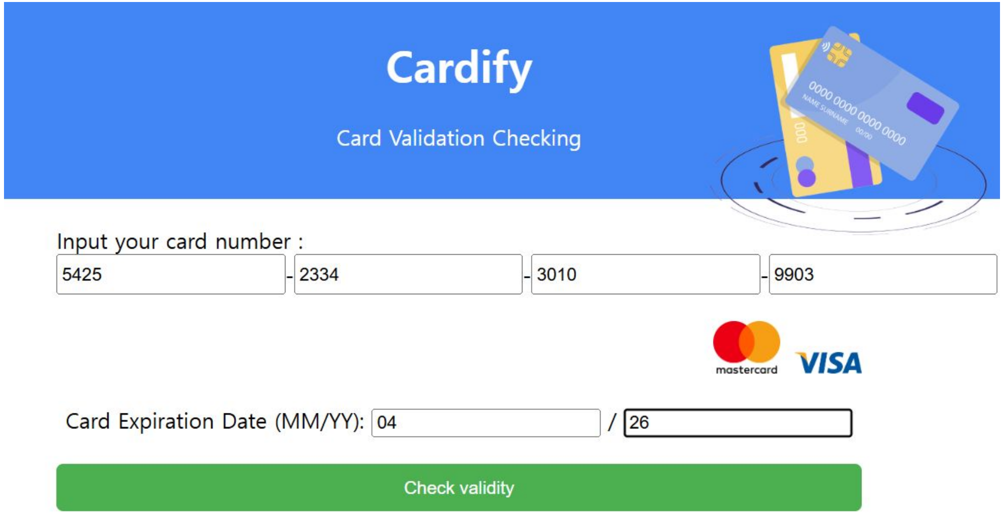
The program will validate the entered card number and expiry date, determine the card brand, and display the validation result on the web page.

Program Execution

INPUT

- **Valid** card number
- **Valid** Expiration date
- Mastercard

RESULT

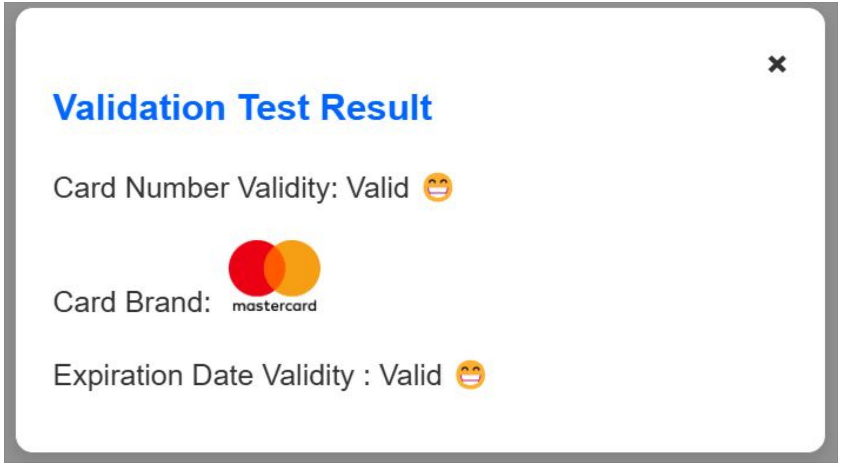


Cardify

Card Validation Checking

Input your card number :
5425 - 2334 - 3010 - 9903

Card Expiration Date (MM/YY): 04 / 26



Validation Test Result

Card Number Validity: Valid 😁

Card Brand:

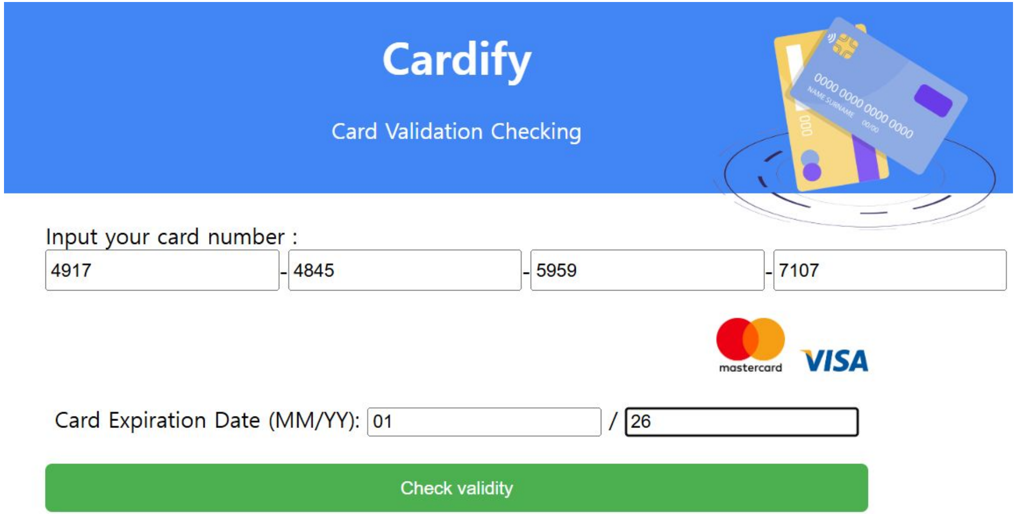
Expiration Date Validity : Valid 😁

Program Execution

INPUT

- **Invalid** card number
- **Valid** Expiration date
- VISA

RESULT

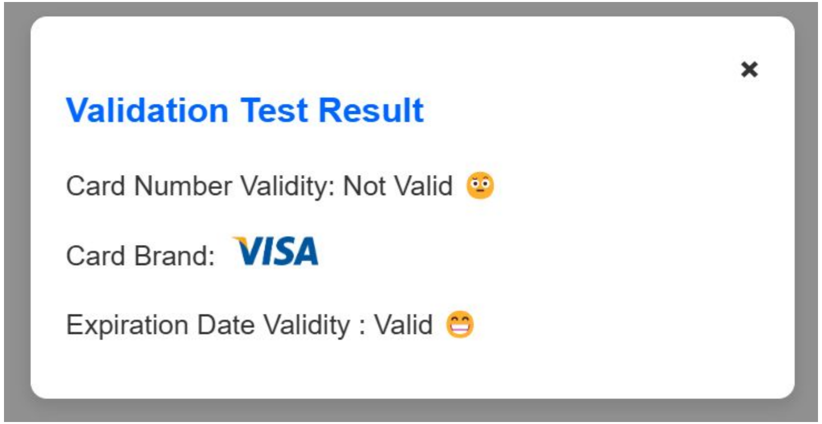


Cardify

Card Validation Checking

Input your card number : 4917 - 4845 - 5959 - 7107

Card Expiration Date (MM/YY): 01 / 26



Validation Test Result

Card Number Validity: Not Valid 😳

Card Brand: VISA

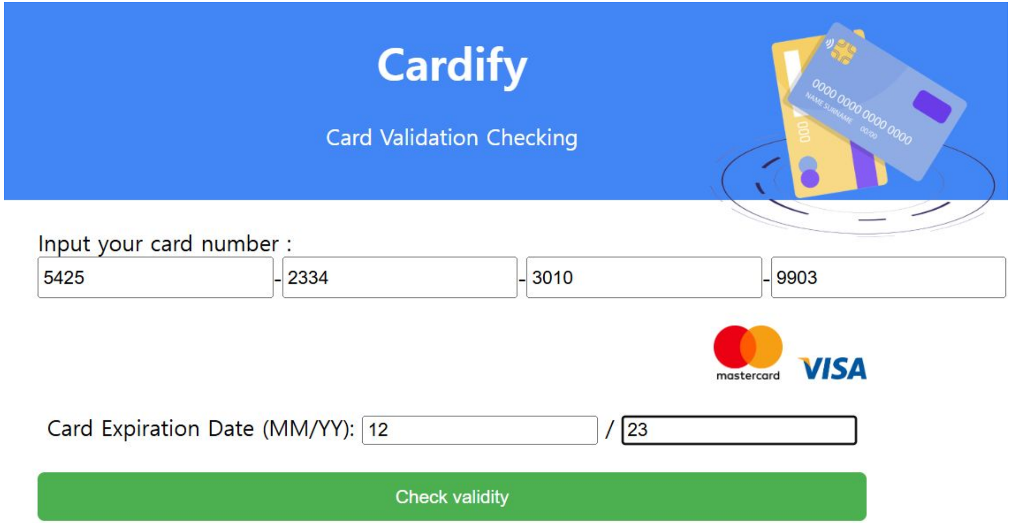
Expiration Date Validity : Valid 😁

Program Execution

INPUT

- **Valid** card number
- **Invalid** Expiration date
- Mastercard

RESULT

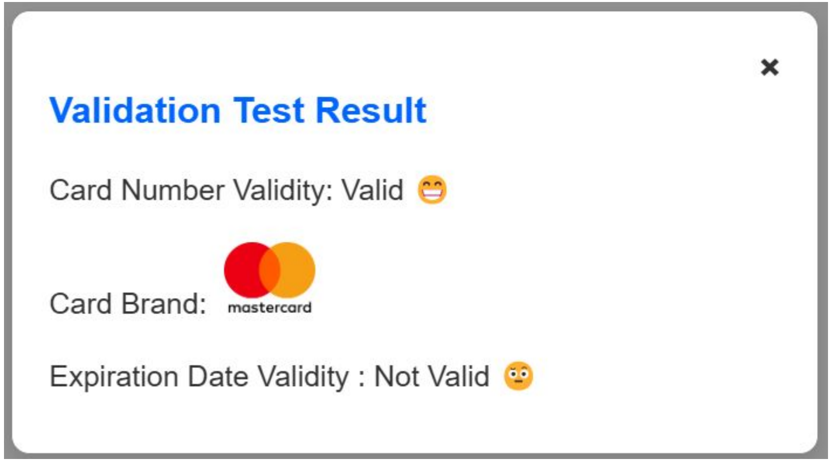


Cardify

Card Validation Checking

Input your card number : 5425 - 2334 - 3010 - 9903

Card Expiration Date (MM/YY): 12 / 23



Validation Test Result

Card Number Validity: Valid 😁

Card Brand: mastercard

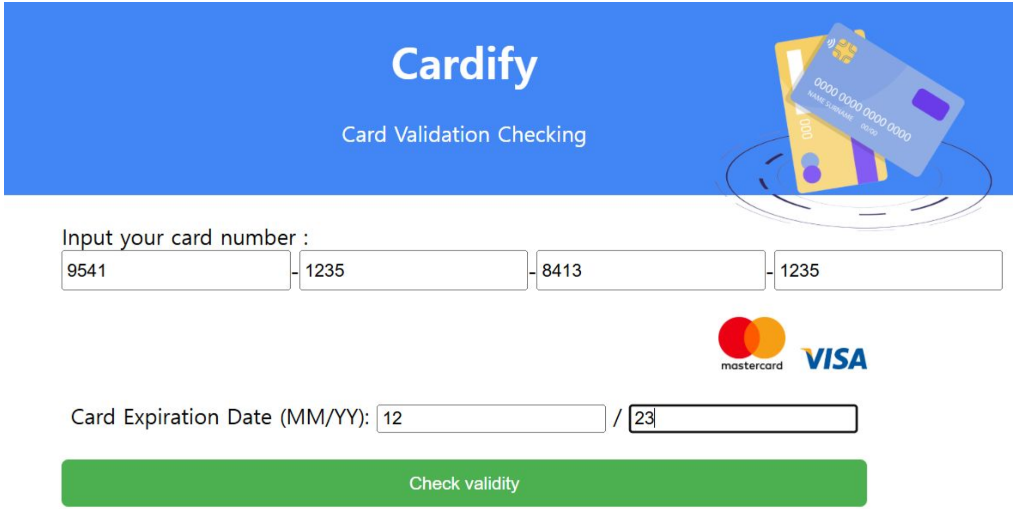
Expiration Date Validity : Not Valid 😳

Program Execution

INPUT

- **Invalid** card number
- **Invalid** Expiration date
- Domestic Card

RESULT



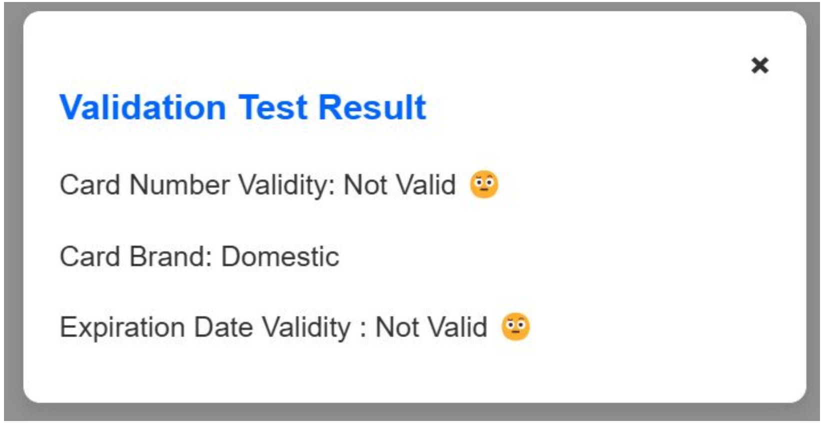
Cardify

Card Validation Checking

Input your card number :

9541 - 1235 - 8413 - 1235

Card Expiration Date (MM/YY): 12 / 23



Validation Test Result

Card Number Validity: Not Valid 😳

Card Brand: Domestic

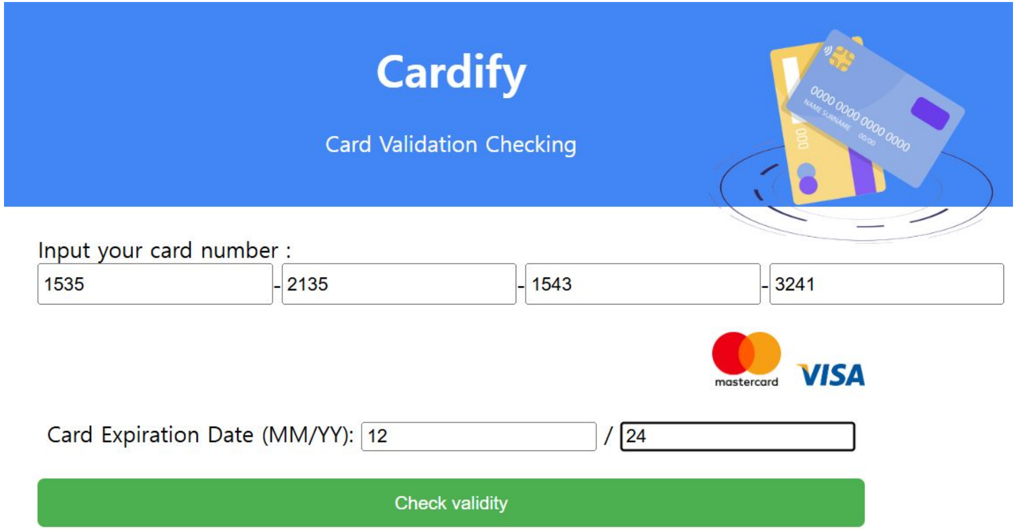
Expiration Date Validity : Not Valid 😳

Program Execution

INPUT

- **Invalid** card number
- **Valid** Expiration date

RESULT

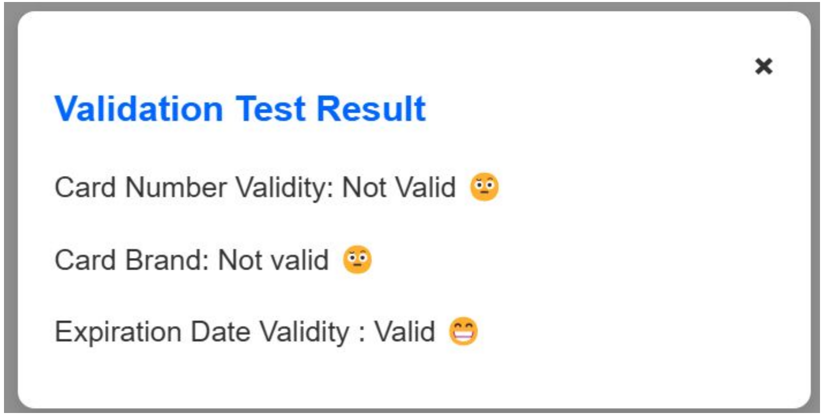


Cardify

Card Validation Checking

Input your card number : 1535 - 2135 - 1543 - 3241

Card Expiration Date (MM/YY): 12 / 24



Validation Test Result

Card Number Validity: Not Valid 😳

Card Brand: Not valid 😳

Expiration Date Validity : Valid 😁

Thank you