**Assignment No : 1**

**Name : Abhishek Shrikrushna Solanke   
Branch : CSE (Artificial Intelligence) - Div (A)  
Roll No : 381013  
PRN : 22310235**

**Title :**

Write a Python program to encrypt and decrypt text using the Play fair Cipher, Vignere Cipher, Simple columnar technique and  Rail fence technique

**Objective :**

1. To understand the working of classical ciphers and their role in cryptography.
2. To implement multiple classical encryption and decryption algorithms.
3. To build a single Python toolkit supporting Playfair, Vigenère, Columnar, and Rail Fence ciphers.
4. To enable both command-line and interactive usage for encrypting and decrypting messages.

**Theory :**

Classical ciphers are the foundation of modern cryptography. They use substitution and transposition techniques to transform plaintext into ciphertext, securing information during transmission.

This project implements four such ciphers:

1. **Playfair Cipher:** Uses a 5x5 matrix of letters derived from a keyword. Pairs of letters are encrypted based on their positions in the matrix.
2. **Vigenère Cipher:** A polyalphabetic substitution cipher that shifts letters using a repeating key.
3. **Columnar Transposition Cipher:** Reorders characters into columns based on a key and reads them in a specific order to produce ciphertext.
4. **Rail Fence Cipher:** Arranges plaintext in a zig-zag pattern across multiple “rails” and reads row by row to encrypt.

These ciphers demonstrate how information can be obscured using mathematical transformations and positional rearrangements, forming the conceptual base for modern cryptographic algorithms.

**Procedure :**

**Steps to Run the Classical Cipher Toolkit:**

1. **Setup Environment**  
   Install Python (≥3.10 recommended).
2. **Save the Program File**  
   Save the following code as classical\_ciphers.py.
3. **Interactive Mode (No arguments):**
4. python classical\_ciphers.py
   1. Choose a cipher (Playfair, Vigenère, Columnar, Rail Fence).
   2. Select operation: Encrypt or Decrypt.
   3. Enter text and key (or rails).
   4. View the result instantly.

**Code :**

"""Assignment 1: Classical cipher toolkit .”””

from \_\_future\_\_ import annotations

import argparse

import itertools

import math

import string

import sys

from typing import Dict, List, Sequence, Tuple

ALPHABET = string.ascii\_uppercase

# Utility helpers

def \_clean\_text(text: str, \*, keep\_space: bool = False) -> str:

cleaned = [ch.upper() for ch in text if ch.isalpha() or (keep\_space and ch.isspace())]

return "".join(cleaned)

def \_chunk\_pairs(text: str) -> List[str]:

pairs: List[str] = []

idx = 0

while idx < len(text):

first = text[idx]

second = text[idx + 1] if idx + 1 < len(text) else "X"

if first == second:

pairs.append(first + "X")

idx += 1

else:

pairs.append(first + second)

idx += 2

if pairs and len(pairs[-1]) == 1:

pairs[-1] += "X"

return pairs

def \_build\_playfair\_matrix(key: str) -> Tuple[List[List[str]], Dict[str, Tuple[int, int]]]:

sanitized = \_clean\_text(key)

seen: Dict[str, bool] = {}

matrix\_letters: List[str] = []

def add\_letter(letter: str) -> None:

letter = "I" if letter == "J" else letter

if letter in seen:

return

seen[letter] = True

matrix\_letters.append(letter)

for char in sanitized:

add\_letter(char)

for char in ALPHABET:

if char == "J":

continue

add\_letter(char)

matrix = [matrix\_letters[i : i + 5] for i in range(0, 25, 5)]

positions: Dict[str, Tuple[int, int]] = {}

for r, row in enumerate(matrix):

for c, char in enumerate(row):

positions[char] = (r, c)

return matrix, positions

def playfair\_encrypt(text: str, key: str) -> str:

matrix, pos = \_build\_playfair\_matrix(key)

pairs = \_chunk\_pairs(\_clean\_text(text))

result: List[str] = []

for first, second in pairs:

r1, c1 = pos[first]

r2, c2 = pos[second]

if r1 == r2:

result.append(matrix[r1][(c1 + 1) % 5] + matrix[r2][(c2 + 1) % 5])

elif c1 == c2:

result.append(matrix[(r1 + 1) % 5][c1] + matrix[(r2 + 1) % 5][c2])

else:

result.append(matrix[r1][c2] + matrix[r2][c1])

return "".join(result)

def playfair\_decrypt(cipher: str, key: str) -> str:

matrix, pos = \_build\_playfair\_matrix(key)

cleaned = \_clean\_text(cipher)

pairs = [cleaned[i : i + 2] for i in range(0, len(cleaned), 2) if len(cleaned[i : i + 2]) == 2]

result: List[str] = []

for first, second in pairs:

r1, c1 = pos[first]

r2, c2 = pos[second]

if r1 == r2:

result.append(matrix[r1][(c1 - 1) % 5] + matrix[r2][(c2 - 1) % 5])

elif c1 == c2:

result.append(matrix[(r1 - 1) % 5][c1] + matrix[(r2 - 1) % 5][c2])

else:

result.append(matrix[r1][c2] + matrix[r2][c1])

return "".join(result)

# Vigenère cipher

def vigenere\_encrypt(text: str, key: str) -> str:

cleaned = \_clean\_text(text, keep\_space=True)

key\_stream = itertools.cycle(\_clean\_text(key))

result: List[str] = []

for ch in cleaned:

if ch.isspace():

result.append(ch)

continue

shift = ord(next(key\_stream)) - ord("A")

offset = (ord(ch) - ord("A") + shift) % 26

result.append(chr(offset + ord("A")))

return "".join(result)

def vigenere\_decrypt(cipher: str, key: str) -> str:

cleaned = \_clean\_text(cipher, keep\_space=True)

key\_stream = itertools.cycle(\_clean\_text(key))

result: List[str] = []

for ch in cleaned:

if ch.isspace():

result.append(ch)

continue

shift = ord(next(key\_stream)) - ord("A")

offset = (ord(ch) - ord("A") - shift) % 26

result.append(chr(offset + ord("A")))

return "".join(result)

# Columnar transposition cipher

def \_columnar\_order(key: str) -> List[int]:

cleaned\_key = \_clean\_text(key)

enumerated = sorted(enumerate(cleaned\_key), key=lambda item: (item[1], item[0]))

order = [None] \* len(cleaned\_key)

for position, (original\_index, \_) in enumerate(enumerated):

order[original\_index] = position

return order # type: ignore

def columnar\_encrypt(text: str, key: str) -> str:

cleaned = \_clean\_text(text, keep\_space=False)

order = \_columnar\_order(key)

width = len(order)

if width == 0:

raise ValueError("Key must contain alphabetic characters")

rows = math.ceil(len(cleaned) / width)

padded = cleaned.ljust(rows \* width, "X")

grid = [padded[i : i + width] for i in range(0, len(padded), width)]

columns: List[str] = [""] \* width

for row in grid:

for idx, char in enumerate(row):

columns[order[idx]] += char

return "".join(columns)

def columnar\_decrypt(cipher: str, key: str) -> str:

order = \_columnar\_order(key)

width = len(order)

if width == 0:

raise ValueError("Key must contain alphabetic characters")

rows = math.ceil(len(cipher) / width)

base\_len = rows \* width

cipher = cipher.ljust(base\_len, "X")

column\_lengths = [rows] \* width

slices: Dict[int, str] = {}

start = 0

for column\_index in range(width):

length = column\_lengths[column\_index]

slices[column\_index] = cipher[start : start + length]

start += length

reordered\_columns = ["" for \_ in range(width)]

for idx, column\_pos in enumerate(order):

reordered\_columns[idx] = slices[column\_pos]

plaintext\_chars: List[str] = []

for row\_idx in range(rows):

for col\_idx in range(width):

plaintext\_chars.append(reordered\_columns[col\_idx][row\_idx])

return "".join(plaintext\_chars).rstrip("X")

# Rail fence cipher

def railfence\_encrypt(text: str, rails: int) -> str:

if rails < 2:

raise ValueError("Rails must be at least 2 for rail fence cipher")

cleaned = \_clean\_text(text, keep\_space=False)

fence = ["" for \_ in range(rails)]

rail = 0

direction = 1

for char in cleaned:

fence[rail] += char

rail += direction

if rail == 0 or rail == rails - 1:

direction \*= -1

return "".join(fence)

def railfence\_decrypt(cipher: str, rails: int) -> str:

if rails < 2:

raise ValueError("Rails must be at least 2 for rail fence cipher")

length = len(cipher)

pattern = \_rail\_pattern(length, rails)

rail\_lengths = [pattern.count(r) for r in range(rails)]

idx = 0

rails\_content: List[List[str]] = []

for count in rail\_lengths:

rails\_content.append(list(cipher[idx : idx + count]))

idx += count

rail\_indices = [0] \* rails

result\_chars: List[str] = []

for rail in pattern:

result\_chars.append(rails\_content[rail][rail\_indices[rail]])

rail\_indices[rail] += 1

return "".join(result\_chars)

def \_rail\_pattern(length: int, rails: int) -> List[int]:

pattern: List[int] = []

rail = 0

direction = 1

for \_ in range(length):

pattern.append(rail)

rail += direction

if rail == 0 or rail == rails - 1:

direction \*= -1

return pattern

# Command-line interface

def \_build\_parser() -> argparse.ArgumentParser:

parser = argparse.ArgumentParser(description="Classical cipher toolkit")

parser.add\_argument("cipher", choices=["playfair", "vigenere", "columnar", "railfence"], help="Cipher to use")

parser.add\_argument("mode", choices=["encrypt", "decrypt"], help="Operation mode")

parser.add\_argument("--text", required=True, help="Plaintext or ciphertext input")

parser.add\_argument("--key", help="Key used by the cipher")

parser.add\_argument("--rails", type=int, help="Number of rails (rail fence only)")

return parser

def \_interactive\_session() -> None:

print("Classical Cipher Toolkit (interactive mode)")

print("Available ciphers:")

options = {

"1": "playfair",

"2": "vigenere",

"3": "columnar",

"4": "railfence",

}

for number, name in options.items():

print(f" {number}. {name.title()}")

cipher\_choice = None

while cipher\_choice not in options:

cipher\_choice = input("Select cipher [1-4]: ").strip()

cipher = options[cipher\_choice]

mode\_choice = None

while mode\_choice not in {"e", "d"}:

mode\_choice = input("Encrypt or decrypt? [e/d]: ").strip().lower()

mode = "encrypt" if mode\_choice == "e" else "decrypt"

text = input("Enter the text: ")

key = None

rails = None

if cipher == "railfence":

while True:

rails\_input = input("Enter number of rails (>=2): ").strip()

try:

rails\_value = int(rails\_input)

except ValueError:

print("Please enter a valid integer.")

continue

if rails\_value < 2:

print("Rails must be at least 2.")

continue

rails = rails\_value

break

else:

while not key:

key = input("Enter the key: ").strip()

if cipher == "playfair":

func = playfair\_encrypt if mode == "encrypt" else playfair\_decrypt

result = func(text, key)

elif cipher == "vigenere":

func = vigenere\_encrypt if mode == "encrypt" else vigenere\_decrypt

result = func(text, key)

elif cipher == "columnar":

func = columnar\_encrypt if mode == "encrypt" else columnar\_decrypt

result = func(text, key)

else:

func = railfence\_encrypt if mode == "encrypt" else railfence\_decrypt

result = func(text, rails)

print("\nResult:")

print(result)

def main(argv: Sequence[str] | None = None) -> None:

if argv is None:

argv = sys.argv[1:]

if not argv:

\_interactive\_session()

return

parser = \_build\_parser()

args = parser.parse\_args(argv)

text = args.text

key = args.key

if args.cipher == "railfence" and args.rails is None:

parser.error("--rails is required for the rail fence cipher")

if args.cipher != "railfence" and (not key):

parser.error("--key is required for this cipher")

if args.cipher == "playfair":

func = playfair\_encrypt if args.mode == "encrypt" else playfair\_decrypt

result = func(text, key)

elif args.cipher == "vigenere":

func = vigenere\_encrypt if args.mode == "encrypt" else vigenere\_decrypt

result = func(text, key)

elif args.cipher == "columnar":

func = columnar\_encrypt if args.mode == "encrypt" else columnar\_decrypt

result = func(text, key)

else: # railfence

func = railfence\_encrypt if args.mode == "encrypt" else railfence\_decrypt

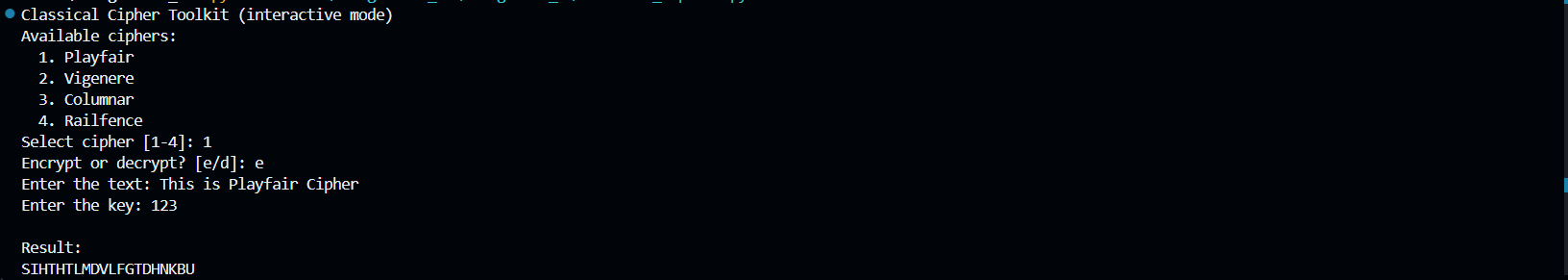
result = func(text, args.rails)

print(result)

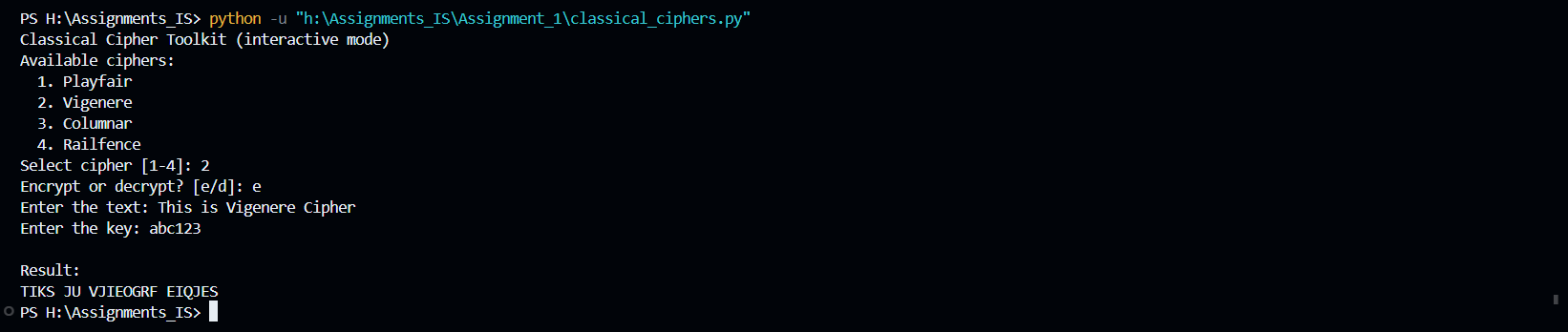
if \_\_name\_\_ == "\_\_main\_\_":

main()

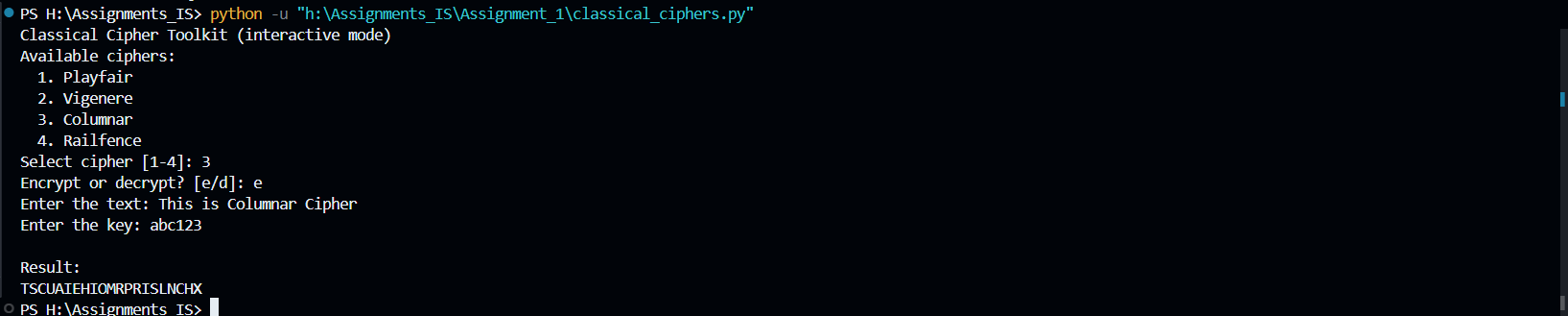
**Output :**

**Example 1 – Playfair Cipher**

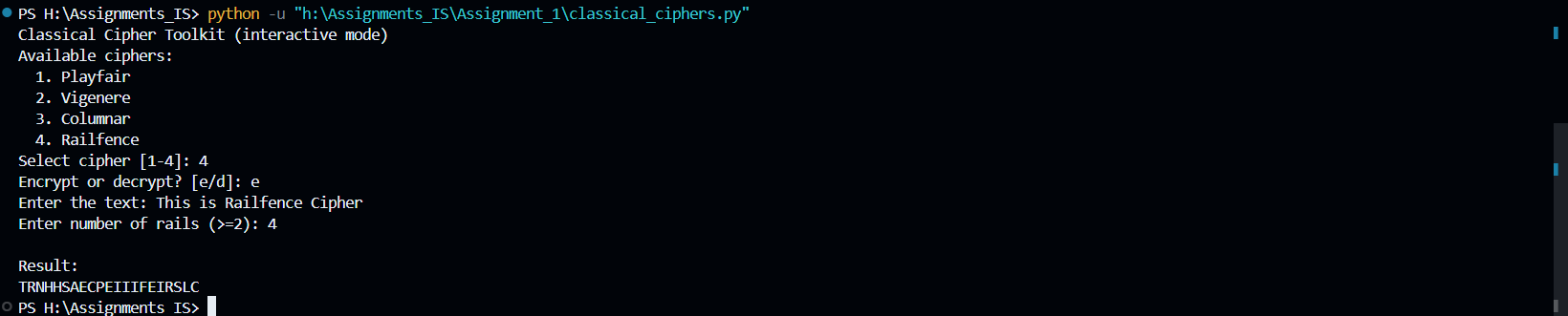
**Example 2 – Vigenère Cipher**



**Example 3 – Columnar Cipher**



**Example 4 – Rail Fence Cipher**



**Conclusion :**

This Classical Cipher code successfully implements multiple encryption and decryption algorithms from early cryptography. By integrating Playfair, Vigenère, Columnar, and Rail Fence ciphers into one Python program, it demonstrates the principles of substitution and transposition techniques.  
This project enhances understanding of manual encryption processes and provides a foundation for studying modern symmetric cryptography.