PYTHON CODES

1. MD5

```
# Python 3 code to demonstrate the
# working of MD5 (byte - byte)

import hashlib

# encoding GeeksforGeeks using md5 hash
# function
result = hashlib.md5(b'GeeksforGeeks')

# printing the equivalent byte value.
print(&quot;The byte equivalent of hash is : &quot;, end =&quot;&quot;)
print(result.digest())
```

2. SHA-1

```
# initializing string
str = "GeeksforGeeks"

# encoding GeeksforGeeks using encode()
# then sending to SHA1()
result = hashlib.sha1(str.encode())

# printing the equivalent hexadecimal value.
print("The hexadecimal equivalent of SHA1 is : ")
print(result.hexdigest())
```

3. Euclidean Algorithm (basic)

```
# Python3 program to demonstrate Basic Euclidean Algorithm
# Function to return gcd of a and b
def gcd(a, b):
    if a == 0:
        return b
    return gcd(b % a, a)
# Driver code
if __name__ == "__main__":
  a = 10
  b = 15
  print("gcd(", a, ",", b, ") = ", gcd(a, b))

  a = 35
  b = 10
  print("gcd(", a, ",", b, ") = ", gcd(a, b))

  a = 31
  b = 2
  print("gcd(", a, ",", b, ") = ", gcd(a, b))
```

3. Euclidean Algorithm (extended)

```
def gcdExtended(a, b):
    # Base Case
    if a == 0:
        return b, 0, 1

    gcd, x1, y1 = gcdExtended(b % a, a)

    # Update x and y using results of recursive
    # call
    x = y1 - (b//a) * x1
    y = x1

    return gcd, x, y


# Driver code
a, b = 35, 15
g, x, y = gcdExtended(a, b)
print("gcd(", a, ",", b, ") = ", g)
```

3. Simple Columnar Transposition

```
def simple_columnar_encrypt(plaintext, key):
    num_columns = len(key)
    num_rows = len(plaintext) // num_columns
    if len(plaintext) % num_columns != 0:
        num_rows += 1
    padded_plaintext = plaintext.ljust(num_rows * num_columns, 'X')
    grid = [padded_plaintext[i::num_rows] for i in range(num_columns)]
    sorted_key = sorted(enumerate(key), key=lambda x: x[1])
    sorted_indices = [index for index, _ in sorted_key]
    ciphertext = ".join(".join(grid[i]) for i in sorted_indices)
    return ciphertext
def simple_columnar_decrypt(ciphertext, key):
    sorted_key = sorted(enumerate(key), key=lambda x: x[1])
    sorted_indices = [index for index, _ in sorted_key]
    num_columns = len(key)
    num_rows = len(ciphertext) // num_columns
    if len(ciphertext) % num_columns != 0:
        num_rows += 1
    columns = [''] * num_columns
    for i, index in enumerate(sorted_indices):
        columns[index] = ciphertext[i * num_rows: (i + 1) * num_rows]
    grid = ['' for _ in range(num_rows)]
    for col in columns:
        for i in range(num_rows):
            if i < len(col):
                grid[i] += col[i]
    decrypted_text = ''.join(grid).rstrip('X')
    return decrypted_text
plaintext = "HELLOTHISISACOLUMNARTRANSPOSITIONEXAMPLE"
key = "KEY"
ciphertext = simple_columnar_encrypt(plaintext, key)
print(f"Ciphertext: {ciphertext}")
decrypted_text = simple_columnar_decrypt(ciphertext, key)
print(f"Decrypted Text: {decrypted_text}")
```

4. Advanced Columnar Transposition

```
import random
import string
def generate_random_key(length):
    """Generate a random string of given length, used for substitution."""
    return ''.join(random.choices(string.ascii_uppercase, k=length))
def advanced_columnar_encrypt(plaintext, key, num_rounds=2):
    num_columns = len(key)
    num_rows = len(plaintext) // num_columns
    if len(plaintext) % num_columns != 0:
        num_rows += 1
    padded_plaintext = plaintext.ljust(num_rows * num_columns, 'X')
    random_key = generate_random_key(len(padded_plaintext))
    substitution_map = {chr(i + 65): random_key[i] for i in range(26)}  # Map for A-Z
    substituted_plaintext = ".join([substitution_map.get(c, c) for c in padded_plaintext])
    grid = [substituted_plaintext[i::num_rows] for i in range(num_columns)]
    for _ in range(num_rounds):
        sorted_key = sorted(enumerate(key), key=lambda x: x[1])
        sorted_indices = [index for index, _ in sorted_key]
        grid = [grid[i] for i in sorted_indices]
    ciphertext = ".join(".join(grid[i]) for i in sorted_indices)
    return ciphertext
def advanced_columnar_decrypt(ciphertext, key, num_rounds=2):
    num_columns = len(key)
    num_rows = len(ciphertext) // num_columns
    if len(ciphertext) % num_columns != 0:
        num_rows += 1
    sorted_key = sorted(enumerate(key), key=lambda x: x[1])
    sorted_indices = [index for index, _ in sorted_key]
    columns = ["] * num_columns
    for i, index in enumerate(sorted_indices):
        columns[index] = ciphertext[i * num_rows: (i + 1) * num_rows]
    grid = [" for _ in range(num_rows)]
    for col in columns:
        for i in range(num_rows):
            if i < len(col):
                grid[i] += col[i]
    for _ in range(num_rounds):
        grid = [grid[i] for i in sorted_indices]
    random_key = generate_random_key(len(ciphertext))
    substitution_map = {random_key[i]: chr(i + 65) for i in range(26)}  # Inverse map for A-Z
    decrypted_text = ".join([substitution_map.get(c, c) for c in ".join(grid)]).rstrip('X')
    return decrypted_text
plaintext = "HELLOTHISISACOLUMNARTRANSPOSITIONEXAMPLE"
key = "KEY"
ciphertext = advanced_columnar_encrypt(plaintext, key)
```

```
print(f"Ciphertext: {ciphertext}")

decrypted_text = advanced_columnar_decrypt(ciphertext, key)

print(f"Decrypted Text: {decrypted_text}")
```

5. Rail Fence algorithm

```
# function to encrypt a message
def encryptRailFence(text, key):
    rail = [['\n' for i in range(len(text))]
                for j in range(key)]
    dir_down = False
    row, col = 0, 0
    for i in range(len(text)):
        if (row == 0) or (row == key - 1):
            dir_down = not dir_down
        rail[row][col] = text[i]
        col += 1
        if dir_down:
            row += 1
        else:
            row -= 1
    result = []
    for i in range(key):
        for j in range(len(text)):
            if rail[i][j] != '\n':
                result.append(rail[i][j])
    return("" . join(result))

def decryptRailFence(cipher, key):
    rail = [['\n' for i in range(len(cipher))]
                for j in range(key)]
    dir_down = None
    row, col = 0, 0
    for i in range(len(cipher)):
        if row == 0:
            dir_down = True
        if row == key - 1:
            dir_down = False
        rail[row][col] = '*'
        col += 1
        if dir_down:
            row += 1
        else:
            row -= 1
    index = 0
    for i in range(key):
        for j in range(len(cipher)):
            if ((rail[i][j] == '*') and
            (index < len(cipher))):
                rail[i][j] = cipher[index]
                index += 1
    result = []
    row, col = 0, 0
    for i in range(len(cipher)):
        if row == 0:
            dir_down = True
        if row == key-1:
            dir_down = False
        if (rail[row][col] != '*'):
            result.append(rail[row][col])
            col += 1
        if dir_down:
            row += 1
        else:
            row -= 1
    return("".join(result))
if __name__ == "__main__":
    print(encryptRailFence("attack at once", 2))
    print(encryptRailFence("GeeksforGeeks ", 3))
    print(encryptRailFence("defend the east wall", 3))

    print(decryptRailFence("GsGsekfrek eoe", 3))
    print(decryptRailFence("atc toctaka ne", 2))
    print(decryptRailFence("dnhaweedtees alf  tl", 3))
```