Prac 1\21430790_Prac_1.py

```
1 # Name: Alexandros Theodorou
   # Student num: 21430790
3
4
   IMPORTANT!!
6
7
   - Due 13 March 2024 (before 8h30)
   - No late submissions (AMS and Turnitin) accepted after 8h30!
   - The prac test starts at 10h30 in the Netlabs.
9
10
   - Rename this file to "<YourStudentNumber>_Prac_1.py", for example: "19056789_Prac_1.py"
11
12
   - Comment your code (follow best practice)
13
   - Submit .py to AMS and a .pdf to ClickUp (TurnItIn)
14
   - Also, please upload your turnitin receipt to the AMS.
15
   - Remove all print statements - and helper functions (that weren't provided) - used for unit testing.
16
17
   - Please read the practical guide for instructions!
18
19
   import string
20
21
   import numpy as np
22
   #from PIL import Image
23
   #import matplotlib.pyplot as plt
24
25
   # 3.1 Playfair Cipher
26
27
   28
29
   def playfair_get_key(isText: bool, key: str) -> np.ndarray: # 3.1.1
30
       if isText:
31
           alph = list(string.ascii_lowercase)
           x=0
32
           new_key = ""
33
34
           while x<len(key):</pre>
35
               if key[x].lower() in alph and key[x].lower() not in new_key: # remove not letters and duplicates
36
                   new_key+=key[x].lower()
37
               x+=1
38
           key list = list(new key)
           key_matrix = key_list
39
40
           y=<mark>0</mark>
41
42
           while len(key_matrix)<25:</pre>
43
               if alph[y] not in key_list and alph[y]!= "j": # fill rest of key matrix
                                                             # with chars not in key or not j
44
                   key_matrix.append(alph[y])
45
               v+=1
46
           key matrix = np.array(key matrix)
47
           key_matrix= key_matrix.reshape(5,5)
48
49
           return key_matrix
       else:
50
           unique_chars = []
51
52
           for char in key:
53
               if char not in unique_chars:
54
                   unique_chars.append(char)
55
56
           ascii_values = [ord(char) % 256 for char in unique_chars]
```

```
57
 58
             all_values = list(range(256))
 59
 60
             for val in ascii_values:
                 if val in all_values:
 61
 62
                     all_values.remove(val)
 63
 64
             key_values = ascii_values + all_values
 65
 66
             key_matrix = np.array(key_values).reshape(16, 16)
 67
 68
             return key_matrix
 69
 70
     def playfair_get_pos_in_key(val, val2, keyMat: np.ndarray) -> np.ndarray: # 3.1.2
 71
         x_val_1 = -1
 72
         y_val_1 = -1
         x_val_2= -1
 73
 74
         y_val_2 = -1
 75
 76
         # Find values in arrays
 77
         for i in range(keyMat.shape[0]):
 78
             for j in range(keyMat.shape[1]):
 79
                 if val == keyMat[i][j]:
 80
                     x_val_1 = i
 81
                     y_val_1 = j
 82
                 if val2 == keyMat[i][j]:
 83
                     x val 2 = i
                     y_val_2 = j
 84
         pos = np.array([x_val_1,y_val_1,x_val_2,y_val_2])
 85
 86
         return pos
 87
 88
     def playfair get encryption pos(pos: np.ndarray, keyMat: np.ndarray) -> np.ndarray: # 3.1.3
 89
         x_len = keyMat.shape[0]
 90
         y_len = keyMat.shape[1]
         # Both vals in same row
 91
 92
         if pos[0] == pos[2]:
 93
             pos[1] = pos[1] + 1 if pos[1] < x_len - 1 else 0
 94
             pos[3] = pos[3] +1 if pos[3] < x_len-1 else 0
 95
             return pos
 96
         # Both vals in same column
 97
         if pos[1] == pos[3]:
 98
             pos[0] = pos[0] +1 if pos[0] < y_len-1 else 0
 99
             pos[2] =pos[2]+1 if pos[2]< y_len-1 else 0
100
             return pos
101
         else:
102
             temp = pos[1]
103
             pos[1] = pos[3]
104
             pos[3] = temp
105
             return pos
106
107
     def playfair_get_decryption_pos(pos: np.ndarray, keyMat: np.ndarray) -> np.ndarray: # 3.1.4
108
         x_len = keyMat.shape[0]
109
         y_len = keyMat.shape[1]
         # Both vals in same row
110
         if pos[0] == pos[2]:
111
             pos[1] = pos[1] -1 if pos[1]!=0 else x_len-1
112
113
             pos[3] = pos[3]-1 if pos[3]!= 0 else x_len-1
114
             return pos
115
         # Both vals in same column
```

```
116
         if pos[1] == pos[3]:
117
             pos[0] = pos[0]-1 if pos[0]!=0 else y_len-1
118
             pos[2] = pos[2]-1 if pos[2]!=0 else y_len-1
119
120
         else:
121
             temp = pos[1]
122
             pos[1] = pos[3]
123
             pos[3] = temp
124
             return pos
125
126
     def playfair_preprocess_text(plaintext: str) -> str: # 3.1.5
127
         alph = list(string.ascii_lowercase)
128
         x=0
         new_text = ""
129
130
         while x<len(plaintext):</pre>
131
             if plaintext[x].lower() in alph:
132
                 if plaintext[x].lower() == "j":
                          new_text+= "i"
133
134
                 else:
135
                          new_text+=plaintext[x].lower()
136
137
             x+=1
138
         y = 0
139
         while y<len(new_text)-2:</pre>
140
             x1 = new_text[y]
             x2 = new_text[y+1]
141
142
             if x1 == x2:
143
                 new_text = new_text[:y+1]+'x' +new_text[y+1:]
144
145
                 y+=2
146
             else:
147
148
                 v+=2
149
         if len(new_text) % 2 !=0:
150
             new_text = new_text+ 'x'
151
         return new_text
152
153
     def playfair_encrypt_text(plaintext: str, key: str) -> str: # 3.1.6
154
155
         new_text = playfair_preprocess_text(plaintext)
156
         key_matrix = playfair_get_key(True,key)
         cipher = ""
157
158
         # Encrypt every two characters
159
         for i in range(0,len(new_text),2):
             x = new_text[i]
160
161
             y = new_text[i+1]
162
             pos= playfair_get_pos_in_key(x,y,key_matrix)
163
             new_pos = playfair_get_encryption_pos(pos,key_matrix)
             cipher = cipher + key_matrix[new_pos[0]][new_pos[1]] + key_matrix[new_pos[2]][new_pos[3]]
164
         return cipher
165
166
167
     def playfair_decrypt_text(ciphertext: str, key: str) -> str: # 3.1.7
168
         #new_text = playfair_preprocess_text(ciphertext)
         key_matrix = playfair_get_key(True,key)
169
170
         plaintext = ""
         # decrypt every two characters
171
172
         for i in range(0,len(ciphertext),2):
173
             x = ciphertext[i]
             y = ciphertext[i+1]
174
```

```
175
             pos= playfair_get_pos_in_key(x,y,key_matrix)
             new_pos = playfair_get_decryption_pos(pos,key_matrix)
176
177
             plaintext = plaintext + key_matrix[new_pos[0]][new_pos[1]] + key_matrix[new_pos[2]][new_pos[3]]
178
179
         x= 0
         new_text = ""
180
181
         while x<len(plaintext):</pre>
182
             if plaintext[x]!= 'x':
183
                 new_text = new_text+ plaintext[x]
184
             x+=1
185
         return new_text
186
187
     def playfair_preprocess_image(plaintext: np.ndarray) -> np.ndarray: # 3.1.8
188
         flattened = plaintext.flatten()
189
         # replace 129 with 128
190
         flattened = np.where(flattened == 129, 128, flattened)
191
192
         if len(flattened) % 2 != 0:
193
             flattened = np.append(flattened, 129)
194
195
         return flattened
196
197
     def playfair_remove_image_padding(plaintextWithPadding: np.ndarray) -> np.ndarray: # 3.1.9
198
         new text = []
199
         for i in range(len(plaintextWithPadding)):
200
             if plaintextWithPadding[i] != 129:
201
                 new_text.append(plaintextWithPadding[i])
202
         return np.array(new_text)
203
204
     def playfair_encrypt_image(plaintext: np.ndarray, key: str) -> np.ndarray: # 3.1.10
205
         processed_image = playfair_preprocess_image(plaintext)
206
         key_matrix = playfair_get_key(False, key)
207
208
         ciphertext = np.zeros_like(processed_image)
209
         # Encrypt two pixels at a time
210
         for i in range(0, len(processed_image), 2):
             if i+1 < len(processed image):</pre>
211
212
                 x = processed_image[i]
213
                 y = processed_image[i+1]
214
215
                 pos = playfair_get_pos_in_key(x, y, key_matrix)
216
217
                 new_pos = playfair_get_encryption_pos(pos, key_matrix)
218
                 ciphertext[i] = key_matrix[new_pos[0]][new_pos[1]]
219
                 ciphertext[i+1] = key_matrix[new_pos[2]][new_pos[3]]
220
221
         return ciphertext
222
223
     def playfair_decrypt_image(removePadding: bool, ciphertext: np.ndarray, key: str) -> np.ndarray: # 3.1.11
224
         key_matrix = playfair_get_key(False, key)
225
         plaintext = np.zeros_like(ciphertext)
226
227
         # Decrypt two pixels at a time
228
         for i in range(0, len(ciphertext), 2):
             if i+1 < len(ciphertext):</pre>
229
230
                 x = ciphertext[i]
231
                 y = ciphertext[i+1]
232
233
                 pos = playfair_get_pos_in_key(x, y, key_matrix)
```

```
234
                 new_pos = playfair_get_decryption_pos(pos, key_matrix)
235
                 plaintext[i] = key_matrix[new_pos[0]][new_pos[1]]
236
                 plaintext[i+1] = key_matrix[new_pos[2]][new_pos[3]]
237
238
         if removePadding:
239
             plaintext = playfair_remove_image_padding(plaintext)
240
241
         return plaintext
242
243
     def playfair_convert_to_image(imageData: np.ndarray, originalShape) -> np.ndarray: # 3.1.12
244
         # Dimentions of image
245
         required_size = originalShape[0] * originalShape[1] * originalShape[2]
         if len(imageData) < required_size:</pre>
246
247
             padding_needed = required_size - len(imageData)
             padded_data = np.pad(imageData, (0, padding_needed), 'constant', constant_values=0)
248
249
             reshaped_image = padded_data.reshape(originalShape)
250
         elif len(imageData) > required_size:
251
             reshaped_image = imageData[:required_size].reshape(originalShape)
252
         else:
253
             reshaped_image = imageData.reshape(originalShape)
254
255
         return reshaped_image
256
257
258
259
     # 3.2 Hill Cipher
260
261
262
     def hill get key(isText: bool, key: str) -> np.ndarray:
263
         if isText:
             key_list = list(key)
264
265
             for i in range(len(key list)):
                 key_list[i] = ord(key_list[i]) % 97
266
             key_list = np.array(key_list)
267
268
             if len(key_list) == 4:
                 key_list = key_list.reshape(2, 2)
269
             if len(key_list) == 9:
270
271
                 key_list = key_list.reshape(3, 3)
272
             det = np.linalg.det(key_list)
273
             if det == 0:
274
                 for i in range(key_list.shape[0]):
275
                     for j in range(key_list.shape[1]):
276
                         key_list[i][j] = -1
277
             return key_list
278
         else:
             # For images, similar process but with modulo 256
279
280
             key_list = list(key)
281
             for i in range(len(key_list)):
                 key_list[i] = ord(key_list[i]) # ASCII value directly, no modulo 97
282
             key_list = np.array(key_list)
283
             if len(key_list) == 4:
284
285
                 key_list = key_list.reshape(2, 2)
286
             elif len(key_list) == 9:
287
                 key_list = key_list.reshape(3, 3)
             else:
288
289
                 return np.array([[-1]]) # Invalid size
290
291
             det = np.linalg.det(key_list) % 256
292
             # Check if determinant has an inverse in modulo 256
```

```
293
             has_inverse = False
294
             for i in range(1, 256):
295
                 if (int(det) * i) % 256 == 1:
296
                     has_inverse = True
297
                     break
298
299
             if det == 0 or not has_inverse:
                 return np.full(key_list.shape, -1)
300
301
302
             return key_list
303
304
     def hill_get_inv_key(isText: bool, keyMat: np.ndarray) -> np.ndarray: # 3.2.2
305
306
         mod = 26 if isText else 256
307
308
         # Calculate the determinant of the key matrix
309
         det = int(round(np.linalg.det(keyMat)))
310
311
         # Ensure the determinant is positive
312
         det = det % mod
313
314
         # Find modular multiplicative inverse of the determinant
315
         det_inv = pow(det, -1, mod)
316
317
         # Calculate the adjugate matrix (transpose of cofactor matrix)
318
         n = keyMat.shape[0]
319
         adj = np.zeros(keyMat.shape, dtype=int)
320
         for i in range(n):
321
322
             for j in range(n):
323
                 # Get the minor by removing row i and column j
                 minor = np.delete(np.delete(keyMat, i, axis=0), j, axis=1)
324
                 # Calculate the cofactor (determinant of minor with sign)
325
326
                 cofactor = round(np.linalg.det(minor))
327
                 # Apply the sign rule (-1)^(i+j)
                 cofactor *= (-1) ** (i + j)
328
                 # Store in the adjugate matrix (transpose of cofactor matrix)
329
330
                 adj[j, i] = cofactor
331
332
         # Calculate the inverse key using the formula: K^{-1} = (\det_{i} v * adj) \mod M
         inv_key = (det_inv * adj) % mod
333
334
335
         return inv_key
336
337
     def hill_process_group(isText: bool, group: np.ndarray, keyMat: np.ndarray) -> np.ndarray: # 3.2.3
         if isText:
338
339
             result = np.dot(keyMat, group) % 26
340
             return result
341
         else:
             result = np.dot(keyMat, group) % 256
342
             return result
343
344
345
     def hill_pre_process_text(plaintext: str, keyLength: int) -> np.ndarray: # 3.2.4
         alph = list(string.ascii_lowercase)
346
347
         new_text =[]
         for s in plaintext:
348
349
             if s.lower() in alph:
350
                 new_text.append(s.lower())
351
         if keyLength == 4:
```

```
352
             while len(new_text) % 2 !=0:
353
                 new_text.append("x")
354
         if keyLength == 9:
355
             while len(new_text) % 3 !=0:
356
                 new_text.append("x")
357
358
         for i in range(len(new_text)):
359
             new_text[i] = ord(new_text[i]) % 97
360
361
         return np.array(new_text)
362
363
     def hill_encrypt_text(plaintext: str, key: str) -> str: # 3.2.5
364
365
366
         key_length = len(key)
367
         key_mat = hill_get_key(True, key)
368
         # Check if key is valid
369
370
         if -1 in key mat:
371
             return "Invalid Key"
372
373
         processed_text = hill_pre_process_text(plaintext, key_length)
374
375
         # Determine the group size based on key length
         group_size = 2 if key_length == 4 else 3
376
377
378
         # Encrypt
379
         result = []
         for i in range(0, len(processed_text), group_size):
380
381
             group = processed_text[i:i+group_size]
382
             encrypted_group = hill_process_group(True, group, key_mat)
383
             # Convert numbers back to characters
384
385
             for num in encrypted_group:
386
                 result.append(chr(num + 97))
387
388
         return "".join(result)
389
390
     def hill_decrypt_text(ciphertext: str, key: str) -> str: # 3.2.6
391
392
         import numpy as np
393
394
         key_length = len(key)
395
         key_mat = hill_get_key(True, key)
396
397
         # Check if key is valid
398
         if -1 in key_mat:
399
             return "Invalid Key"
400
401
         inv_key_mat = hill_get_inv_key(True, key_mat)
402
403
         processed_text = []
404
         for char in ciphertext:
405
             processed_text.append(ord(char.lower()) - 97)
406
         processed_text = np.array(processed_text)
407
408
         # Determine the group size based on key length
409
         group_size = 2 if key_length == 4 else 3
410
```

```
411
         # Decrypt
412
         result = []
413
         for i in range(0, len(processed_text), group_size):
414
             group = processed_text[i:i+group_size]
415
             decrypted_group = hill_process_group(True, group, inv_key_mat)
416
417
             # Convert numbers back to characters
418
             for num in decrypted_group:
419
                 result.append(chr(num + 97))
420
421
         # remove padding
         plaintext = "".join(result)
422
423
         while plaintext.endswith('x'):
424
             plaintext = plaintext[:-1]
425
426
         return plaintext
427
     def hill_pre_process_image(plaintext: np.ndarray, keyLength: int) -> np.ndarray: # 3.2.7
428
429
430
         flattened = plaintext.flatten()
431
432
         flattened = np.where(flattened == 129, 128, flattened)
433
434
         # Determine the group size
435
         group_size = 2 if keyLength == 4 else 3
436
437
         # Add padding if necessary
         padding_needed = (group_size - (len(flattened) % group_size)) % group_size
438
439
         if padding_needed > 0:
440
             padding = np.full(padding_needed, 129)
441
             flattened = np.concatenate((flattened, padding))
442
         return flattened
443
444
445
     def hill_encrypt_image(plaintext: np.ndarray, key: str) -> np.ndarray: # 3.2.8
446
447
448
         # Get the key matrix
449
         key_length = len(key)
450
         key_mat = hill_get_key(False, key)
451
452
         # Pre-process the image
453
         processed_image = hill_pre_process_image(plaintext, key_length)
454
455
         # Determine the group size based on key length
456
         group_size = 2 if key_length == 4 else 3
457
458
         # Encrypt
459
         result = np.zeros_like(processed_image)
         for i in range(0, len(processed_image), group_size):
460
461
             group = processed_image[i:i+group_size]
462
             encrypted_group = hill_process_group(False, group, key_mat)
463
             result[i:i+group_size] = encrypted_group
464
         return result
465
466
467
     def hill_decrypt_image(ciphertext: np.ndarray, key: str) -> np.ndarray: # 3.2.9
468
469
```

```
470
471
         # Get the key matrix
472
         key_length = len(key)
473
         key_mat = hill_get_key(False, key)
474
475
         # Get inverse key matrix
476
         inv_key_mat = hill_get_inv_key(False, key_mat)
477
478
         # Determine the group size based on key length
         group_size = 2 if key_length == 4 else 3
479
480
481
         # Decrypt
482
         result = np.zeros_like(ciphertext)
         for i in range(0, len(ciphertext), group_size):
483
484
             group = ciphertext[i:i+group_size]
485
             decrypted_group = hill_process_group(False, group, inv_key_mat)
486
             result[i:i+group_size] = decrypted_group
487
488
         # Remove padding
         result = result[result != 129]
489
490
491
         return result
492
     def hill_convert_to_image(imageData: np.ndarray, originalShape: tuple) -> np.ndarray: # 3.2.10
493
494
495
496
         # Calculate the size needed for the original shape
         original_size = originalShape[0] * originalShape[1] * originalShape[2]
497
498
499
         # Pad where needed
         if len(imageData) < original_size:</pre>
500
             padding needed = original size - len(imageData)
501
             imageData = np.concatenate((imageData, np.zeros(padding_needed)))
502
503
504
         # In case the decoded data is longer than the original
         if len(imageData) > original_size:
505
             imageData = imageData[:original_size]
506
507
508
         # Reshape the data to the original shape
509
         reshaped_image = imageData.reshape(originalShape)
510
511
         return reshaped image
512
513
514
515
    # 3.3 Row Transposition Cipher
516
517
518
     def row_gen_key(key: str) -> np.ndarray: # 3.3.1
519
         ascii_key = []
520
         new_key = []
521
         key_matrix = []
522
         # get rid of duplicates
523
         for x in key:
524
525
             if x not in new_key:
526
                 new_key.append(x)
527
         # convert to ascii
         for i in new_key:
528
```

```
529
             ascii_key.append(ord(i))
530
531
         sorted_ascii = list(ascii_key)
532
         sorted_ascii.sort() # sorted ascendingly to determine order
533
         for i in sorted_ascii:
534
             key_matrix.append(ascii_key.index(i))
535
536
         return np.array(key_matrix)
537
538
     def row_pad_text(plaintext: str, key: np.ndarray) -> str: # 3.3.2
539
         length = len(key)
540
         while len(plaintext) % length !=0:
541
             plaintext = plaintext+ 'x'
542
         return plaintext
543
544
     def row_encrypt_single_stage(plaintext: str, key: np.ndarray) -> str:
545
         plaintext = row_pad_text(plaintext, key)
546
         # dimensions of block that I am seperating it into
547
         key length = len(key)
548
         num_rows = len(plaintext) // key_length
549
         ciphertext = ""
550
551
         for k in key: # each number of the key
552
             for i in range(num rows):
553
                 position = i * key_length + k # i* key_length controls row count, k then indexes to
554
                                                # correct cipher
555
                 ciphertext += plaintext[position]
556
         return ciphertext
557
558
559
     def row_decrypt_single_stage(ciphertext: str, key: np.ndarray) -> str:
         key length = len(key)
560
561
         num_rows = len(ciphertext) // key_length
562
563
         # create an empty matrix to house the decrypted text
         matrix = [[''] * key_length for _ in range(num_rows)]
564
565
566
         # go column by column for decrypion into the matrix
567
         char_index = 0
568
         for k_index, k_value in enumerate(key):
569
             for i in range(num_rows):
                 matrix[i][k_value] = ciphertext[char_index]
570
571
                 char_index += 1
572
573
         # move lists of characters into the plaintext
         plaintext = ""
574
575
         for row in matrix:
576
             plaintext += ''.join(row)
577
578
         return plaintext
579
580
     def row_encrypt(plaintext: str, key: str, stage: int) -> str:
581
         key_array = row_gen_key(key)
582
         ciphertext = row_encrypt_single_stage(plaintext, key_array)
583
584
585
         # second stage if required
586
         if stage > 1:
587
             ciphertext = row_encrypt_single_stage(ciphertext, key_array)
```

```
588
        return ciphertext
589
590
    def row_decrypt(ciphertext: str, key: str, stage: int) -> str:
591
592
593
         key_array = row_gen_key(key)
594
595
        plaintext = row_decrypt_single_stage(ciphertext, key_array)
596
        # Second stage if required
597
598
        if stage > 1:
599
            plaintext = row_decrypt_single_stage(plaintext, key_array)
600
        # Remove padding
601
        plaintext = plaintext.rstrip('x')
602
603
604
         return plaintext
605
606
607
```