

Prac 1\21430790_Prac_1.py

```
1 # Name: Alexandros Theodorou
2 # Student num: 21430790
3
4 '''
5 IMPORTANT!!
6
7 - Due 13 March 2024 (before 8h30)
8 - No late submissions (AMS and Turnitin) accepted after 8h30!
9 - The prac test starts at 10h30 in the Netlabs.
10
11 - Rename this file to "<YourStudentNumber>_Prac_1.py", for example: "19056789_Prac_1.py"
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
20 import string
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)
32         x=0
33         new_key = ""
34         while x<len(key):
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)
39         key_matrix = key_list
40
41         y=0
42         while len(key_matrix)<25:
43             if alph[y] not in key_list and alph[y]!= "j": # fill rest of key matrix
44                 key_matrix.append(alph[y]) # with chars not in key or not j
45             y+=1
46         key_matrix = np.array(key_matrix)
47         key_matrix= key_matrix.reshape(5,5)
48
49         return key_matrix
50     else:
51         unique_chars = []
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
59     all_values = list(range(256))
60     for val in ascii_values:
61         if val in all_values:
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
73     x_val_2 = -1
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
84                 y_val_2 = j
85     pos = np.array([x_val_1, y_val_1, x_val_2, y_val_2])
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]
91     # Both vals in same row
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]
110     # Both vals in same row
111     if pos[0] == pos[2]:
112         pos[1] = pos[1] - 1 if pos[1] != 0 else x_len - 1
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         return pos
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
129     new_text = ""
130     while x<len(plaintext):
131         if plaintext[x].lower() in alph:
132             if plaintext[x].lower() == "j":
133                 new_text+= "i"
134             else:
135                 new_text+=plaintext[x].lower()
136
137         x+=1
138     y = 0
139     while y<len(new_text)-2:
140         x1 = new_text[y]
141         x2 = new_text[y+1]
142
143         if x1 == x2:
144             new_text = new_text[:y+1]+'x' +new_text[y+1:]
145             y+=2
146
147         else:
148             y+=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)
157     cipher = ""
158     # Encrypt every two characters
159     for i in range(0,len(new_text),2):
160         x = new_text[i]
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)
164         cipher = cipher + key_matrix[new_pos[0]][new_pos[1]] + key_matrix[new_pos[2]][new_pos[3]]
165     return cipher
166
167 def playfair_decrypt_text(ciphertext: str, key: str) -> str: # 3.1.7
168     #new_text = playfair_preprocess_text(ciphertext)
169     key_matrix = playfair_get_key(True,key)
170     plaintext = ""
171     # decrypt every two characters
172     for i in range(0,len(ciphertext),2):
173         x = ciphertext[i]
174         y = ciphertext[i+1]
```

```
175     pos= playfair_get_pos_in_key(x,y,key_matrix)
176     new_pos = playfair_get_decryption_pos(pos,key_matrix)
177     plaintext = plaintext + key_matrix[new_pos[0]][new_pos[1]] + key_matrix[new_pos[2]][new_pos[3]]
178
179     x= 0
180     new_text = ""
181     while x<len(plaintext):
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):
211         if i+1 < len(processed_image):
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):
229         if i+1 < len(ciphertext):
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]
246     if len(imageData) < required_size:
247         padding_needed = required_size - len(imageData)
248         padded_data = np.pad(imageData, (0, padding_needed), 'constant', constant_values=0)
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:
264         key_list = list(key)
265         for i in range(len(key_list)):
266             key_list[i] = ord(key_list[i]) % 97
267         key_list = np.array(key_list)
268         if len(key_list) == 4:
269             key_list = key_list.reshape(2, 2)
270         elif len(key_list) == 9:
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:
279         # For images, similar process but with modulo 256
280         key_list = list(key)
281         for i in range(len(key_list)):
282             key_list[i] = ord(key_list[i]) # ASCII value directly, no modulo 97
283         key_list = np.array(key_list)
284         if len(key_list) == 4:
285             key_list = key_list.reshape(2, 2)
286         elif len(key_list) == 9:
287             key_list = key_list.reshape(3, 3)
288         else:
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:
300         return np.full(key_list.shape, -1)
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
318     n = keyMat.shape[0]
319     adj = np.zeros(keyMat.shape, dtype=int)
320
321     for i in range(n):
322         for j in range(n):
323             minor = np.delete(np.delete(keyMat, i, axis=0), j, axis=1)
324             cofactor = round(np.linalg.det(minor))
325             cofactor *= (-1) ** (i + j)
326             adj[j, i] = cofactor
327
328     # Calculate the inverse key
329     inv_key = (det_inv * adj) % mod
330
331     return inv_key
332
333 def hill_process_group(isText: bool, group: np.ndarray, keyMat: np.ndarray) -> np.ndarray: # 3.2.3
334     if isText:
335         result = np.dot(keyMat, group) % 26
336         return result
337     else:
338         result = np.dot(keyMat, group) % 256
339         return result
340
341 def hill_pre_process_text(plaintext: str, keyLength: int) -> np.ndarray: # 3.2.4
342     alph = list(string.ascii_lowercase)
343     new_text = []
344     for s in plaintext:
345         if s.lower() in alph:
346             new_text.append(s.lower())
347     if keyLength == 4:
348         while len(new_text) % 2 != 0:
349             new_text.append("x")
350     if keyLength == 9:
351         while len(new_text) % 3 != 0:
```

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

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



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

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

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