Final Project: Cryptography and Codebreaking

1 Introduction

Cybersecurity

Let's start by learning a little bit about the world of cybersecurity. The following explanation is adapted from Carrie Anne Philbin's Crash Course Computer Science videos, which I definitely recommend if you are interested in the world of computer science beyond Python!

Cybersecurity is a set of techniques to protect the secrecy, integrity, and availability of computer systems and data against threats. Just as we have physical locks, fences, alarm systems, security guards, etc. to protect physical things from threats, cybersecurity protects digital things from threats.

One way this works is through authentication, the process by which a computer understands who it's interacting with. This could be through a password, biometrics like a fingerprint reader, or something physical like your phone. When you set up Duo to log into your MIT Kerberos account, you were using multifactor authentication, which is even more secure than using just one type.

The opponent to cybersecurity is hackers or cyberattackers. Hackers use their technical knowledge to break into computer systems. 'White hats' are hired to evaluate security, while 'black hats' have malicious intent. Cyberattacks are a huge problem - in fact, they cost the global economy roughly half a trillion dollars annually! Cybersecurity and all of its related fields - data loss prevention, incident response, etc. have been and continue to be vital to our ability to safely use our computers and the internet.

Unfortunately, there is no such thing as a perfect 100% secure computer system. By layering security systems and encrypting our data through cryptography, we can make it harder for hackers to access our information.

What is cryptography?

'Crypto' means hidden or secret, and 'graph' means to write. So cryptography is the process of secret writing.

How do we make writing secret? We use a cipher, an algorithm that converts plain text into cipher text, which is unreadable and makes no sense. Encryption is the process of making text secret, while decryption is the reverse.

Ciphers were actually used way before computers were invented, as far back as 1500 BC in Mesopotamia. Modern cryptology is thought to have originated among Arabs around AD 800. Mechanical and electromechanical cipher machines were in wide use by World War II, and in the US and the UK women were large contributors to code breaking efforts during the war. Today, the way we share keys and messages has changed with the internet, but the basic principles of cryptography stays the same.

Stop and think: What are the pros and cons of using the internet to communicate with people who you can't meet in person? Are messages and keys more or less secure than they were 1000 years ago?

The project

Over the course of this project, you will learn about two main types of ciphers: substitution and transposition. You will learn how to implement the encryption and decryption of specific examples of these, and code a hacking program to break them!

In addition to what you already have learned in WTP, this project will broaden your Python skills. We will use methods from the random, time, and sys modules and learn how to read and write files in Python.

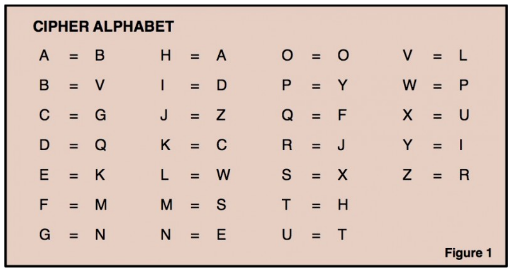
This project is adapted from Al Sweigart's book Cracking Codes with Python. If you enjoy this project and want to continue learning about ciphers, I recommend buying the book and continuing from chapter 13!

"This book is for those who are curious about encryption, hacking, or cryptography. The ciphers in this book [are all centuries old] but by learning them, you'll learn the foundations cryptography was built on and how hackers can break weak encryption."*Al Swiegart, Cracking Codes with Python*

2 Caesar Cipher

How it works

The Caesar cipher is one of the most well known ciphers, developed around 100 BC by Julius Caesar to protect military messages. It is the simplest version of a substitution cipher, in which every character in the plaintext message is changed to another character. To encrypt a message, you follow the cipher key to change each character, for example they may change all 'A's to 'B's and so on. To decrypt the message, the receiver also needs the key and simply does the same process in reverse, turning the 'B's back into 'A's etc.

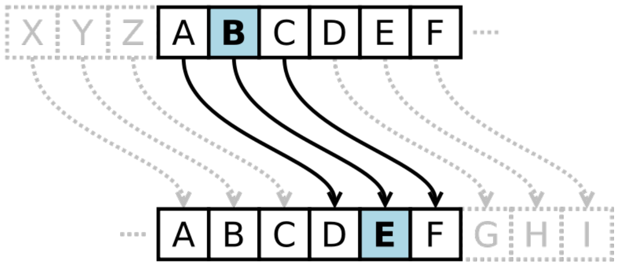


CIPHER ALPHABET

A = B	H = A	O = O	V = L
B = V	I = D	P = Y	W = P
C = G	J = Z	Q = F	X = U
D = Q	K = C	R = J	Y = I
E = K	L = W	S = X	Z = R
F = M	M = S	T = H	
G = N	N = E	U = T	

Figure 1

Generic substitution cipher example



Caesar cipher with shift of 3

The Caesar cipher, more specifically, is a monoalphabetic substitution cipher, in which the alphabet is shifted by some integer in order to determine the key. Caesar famously used a 3-shift cipher: so 'A's become 'D's, 'B's become 'E's, and so on. The last letters wrap around to the beginning of the alphabet, so 'X's become 'A's, etc.

Try it! You can easily implement a Caesar cipher by hand using a cipher wheel. It may be easier to start here so you fully understand how the cipher works before jumping into the code.

Encrypting and decrypting

Open the file caesar.py. Follow along, read the comments and fill in the code. Feel free to add your own comments to understand your code better!

1. We'll start with a very basic Caesar cipher and then make it slightly more complex. **Create a function that translates plaintext into ciphertext according to some key.** We'll assume the shift is to the right, i.e. a shift of three would move 'A' to 'D', etc. Things to keep in mind:
   - What are the possible key values?
   - How do you deal with shifts that wraparound to the start of the alphabet?
   - How do you deal with symbols that aren't in the alphabet?
2. Now test the function with the two test messages, or feel free to make up your own! Does your function work as expected? Do you think the messages are sufficiently concealed?

   As we can see, when we implement the Caesar cipher with only the lowercase letters on our 'wheel', uppercase letters and any other symbols - such as the street number and money signs in the example above - are unchanged, making the majority of our secret message not so secret! To fix this, we can expand our wheel to include more symbols. This might mean letters get shifted to symbols and vice versa, and our end message will be more sufficiently encrypted.
3. Using `symbols` as your 'wheel', implement a Caesar encryption and decryption process. You may make these separate functions, or one function with a parameter `mode`. Think about these questions: How can you reuse code for encrypting and decrypting? What changes between the two modes?
4. Finally, test your completed Caesar cipher with a few different messages and keys. You can choose keys or use `random.randrange()` to choose a random key (make sure the `random` module is imported)

Hacking

Before we hack the Caesar cipher you implemented, let's talk a little bit about algorithms. An algorithm is a finite set of instructions that accomplish a specific task. The recipe for your favorite dessert, Siri's driving directions, and a crochet pattern are all algorithms. Computers also understand algorithms, as long as we write them in a language they know such as Python. Algorithms are made up of three things:

1. Sequencing: applying each step in the order the statements are given. Python reads our code and executes it line by line.
2. Iteration: repeating a portion of an algorithm. We accomplish this in python through for and while loops.
3. Selection: determining what portions of an algorithm are executed based on conditions. This is conditionals, or if/elif/else statements in Python.

Let's explore a few different types of algorithms (by no means an exhaustive list):

- Divide and conquer: Divide problem into smaller similar subproblems and solve those first
- Brute force: Try all possible solutions until you find one that works
- Recursive: Solve the simplest version of the problem, then increase the size
- Dynamic programming: Break into simpler subproblems and solve each problem only once, storing the solution to re-use

The above are all really interesting to learn more about, but for this project we are going to focus on arguably the simplest: brute force. Brute force is the "trial and error" of algorithms. It's the equivalent of trying every path in a maze until you find the exit, or typing every possible PIN number to try and break into your friend's phone.

Stop and think: Are brute force algorithms efficient? How would we determine whether an algorithm is efficient or not?

In the cybersecurity world, brute force attacks can be dangerous. Computers can make millions of guesses to passwords or cipher keys per second, so trying every possibility isn't unreasonable. There are different ways to protect against this, such as limiting the number of tries (have you ever been locked out of your phone?) or adding additional forms of authentication.

However, one of the best ways is simply to increase the number of possible keys a hacker would have to try. Standard RSA encryption, for example, uses keys that are 2048 bits long, meaning there are 2^{2048} or 3.23E616 possibilities! It would take a typical computer around 300 trillion years to crack the code.

Our Caesar cipher code only has as many possible keys as there are symbols on our wheel, making it very easy to hack. Let's try it now!

Open the file bruteforce.py. Fill in the function and test your code by encrypting a few of your own messages. Then, reveal the secret messages.

You'll notice that there is a line that says `import caesar` at the top. In the past, we have imported modules such as random or math, but you can actually import your own code into other Python files! There are two ways to import something into a file:

- Let's say we have a class called `Thing` in a file called **thing.py**. If we want to import this `Thing` into another file, I can use:

  ```python
  from thing import Thing

  t = Thing()
  ```

  This would successfully create an instance of `Thing` and assign it to `t`. We will use this in future sections after we create a class.

- We can also import the entire python file, and then we need to reference the file name anytime we want to use something from the code, with the syntax of classes.

  ```
  import thing

  t = thing.Thing()
  ```

  Here, we are doing the same thing as above, but just importing the entire file instead of the class only.

Stop and think: These are both two valid ways to import code from other files. What are the pros and cons of using each technique?

So we've seen how easy it is to use a brute force algorithm to hack a Caesar cipher. The weakness is that there are just not that many symbols, and therefore, not many possible keys. To make a more secure cipher, we need a cipher that has more possible keys so it takes longer to try every one and look through the results for the correct decryption. One way to accomplish this is with a transposition cipher.

3 Transposition Cipher

What is a transposition cipher?

While a substitution cipher substitutes each character in the message with a different character, a transposition cipher keeps the original characters but rearranges or permutes them into a scrambled mess. There is order to the way they are scrambled, so someone with the key can unscramble them into the original message.

There are different types of transposition ciphers:

- rail fence cipher
- route cipher
- Myszkowski transposition cipher
- disrupted transposition cipher
- columnar transposition cipher

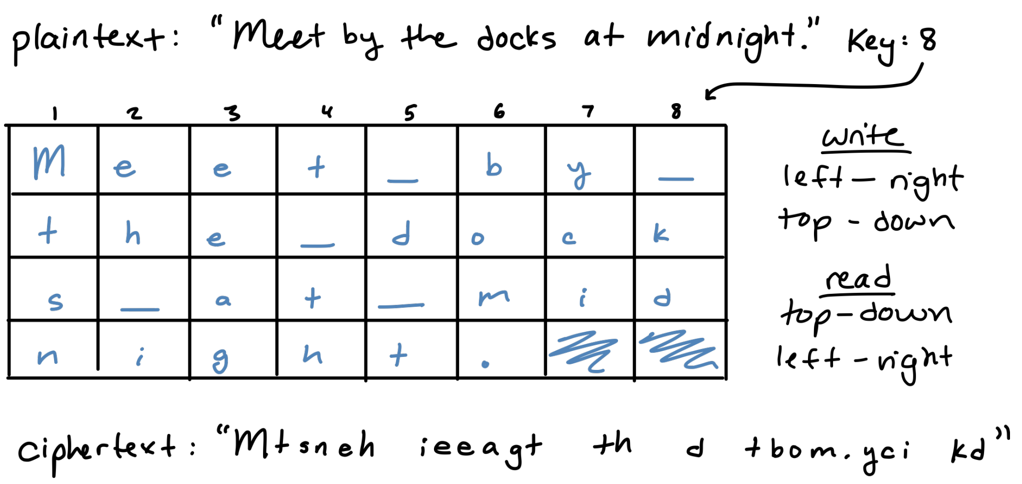
We will implement a columnar transposition cipher in this section.

How does a columnar transposition cipher work?

To encrypt a message with a columnar transposition cipher, you follow these steps:

1. Choose a integer key (at least 2 and no more than half the length of the plaintext message)
2. Create a row of boxes with number of columns equal to the key
3. Fill in the boxes left to right with one character per box (including spaces!). Add more rows as needed (so the entire message should read left to right, top to bottom)
4. On the last row, shade any boxes you didn't use.
5. Read the characters starting from the top left and going down each column, moving to the next column when you reach the bottom. Skip any shaded boxes. This is the ciphertext!

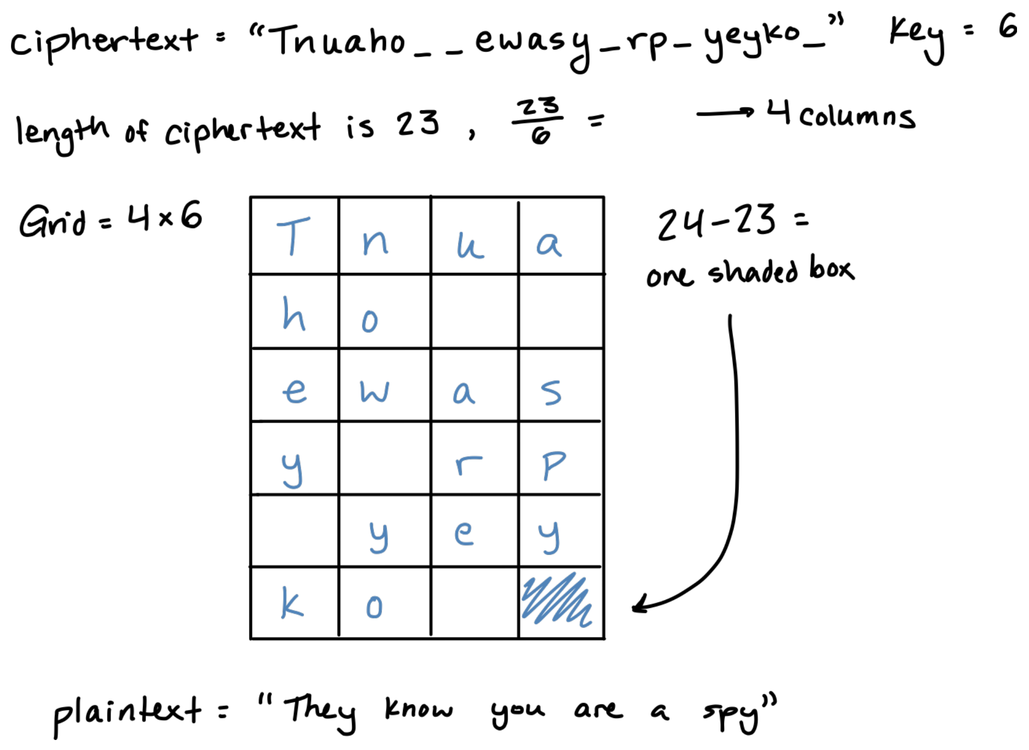
Here's an example:


plaintext: "Meet by the docks at midnight." Key: 8
1 2 3 4 5 6 7 8
M e e t _ b y _
t h e _ d o c k
s _ a t _ m i d
n i g h t .
write left – right top - down
read top – down left - right
ciphertext: "Mtsneh ieeagt th d tbom.yci kd"

To decrypt the ciphertext, we follow a similar process but in reverse:

1. Calculate the number of columns you need by dividing the length of the message by the key and rounding up. Draw a grid with this many columns and as many rows as the key.
2. Subtract the length of the ciphertext from the total number of boxes to figure out how many need to be shaded in. Shade in this many on the bottom of the last column.
3. Fill in the cipher text left to right, top to bottom as before. Skip any shaded boxes.
4. Finally, read the decrypted message down each column, from the leftmost column to the right. This is the plaintext message!

Here's an example of decrypting a mystery message:


ciphertext = "Tnuaho__ewasy_rp_yeyko_" Key = 6
length of ciphertext is 23, 23/6 = → 4 columns
Grid = 4×6
T n u a
h o
e w a s
y r p
y e y
k o
24–23 = one shaded box
plaintext = "They know you are a spy"

Try it! Write out a short message and encrypt/decrypt it using a columnar transposition cipher. Practice this skill a few times to ensure you have a good grasp of the concept before attempting to implement it in Python!

Encrypting and decrypting

Now let's try to implement the above processes in Python!

Open the file transposition.py. We are creating a class `TransCipher`. Instances of the class are a message/key combination, and both the plaintext and ciphertext versions of the message can be stored in the object. When we instantiate an instance of `TransCipher`, we input a `message`, `key`, and boolean `isplaintext` explaining if the message is in plaintext (`True`) or ciphertext (`False`). Then we can use the methods within the class to encrypt or decrypt.
Your task is to fill in the methods in the class `TransCipher` and test them.

Things to think about:

- How to represent the grid of characters
- How to avoid "List index out of range" errors (watch for shaded boxes!)
- Is there a pattern to the indices of characters in each column? Try numbering the characters of a message and encrypting it on paper

Hint: Simulate the columns and rows of the grid by using a list of strings. Each column is represented by one string, where the first character is at the top, etc. At the end of your function, use the `.join()` method to combine your list into one string.

I encourage you to spend the time working out a plan for these methods on your own, even if it doesn't end up being the most efficient! However, if you're really stuck, you can try working from the following pseudocode:

```
def encrypt(self):
# Create a number of columns according to the key.
# Loop through the columns
# Calculate which character from the message would be in your current box (writing left
↪  to right, top to bottom)
# Add the character to the current column
# Go down a row to the next box
# When you get to the bottom of a column, move to the next column on the right and start
↪  at the top row

def decrypt(self):
# Calculate the number of columns you need (the number of rows is the same as the key)
# Calculate the number of boxes to shade in (at the bottom of the furthest right column)
# Write the message in ciphertext left to right, skipping shaded boxes
# Read the plaintext top to bottom, column by column
```

Don't forget to test your code a couple of times to make sure it is working the way you expect! Though, how do you know it will always work? We need to test our methods with a bunch of different messages and keys, but typing all that out sounds like a lot of work. Our first task in the next section is to create an automated testing program that can try a ton of random messages and keys very quickly, so we can be more sure our transposition cipher works perfectly.

4 Some Useful Programs

Testing your code

To test the class we wrote in the previous section, we want to 1) create a random message and random key, 2) encrypt it, 3) decrypt it, and 4) see if the final product is the same as the message we started with. Then we want to do this a bunch of times.

Open the file transposition_test.py. Fill in the code according to the comments then use it to run many (at least 50) encryption/decryption tests in a short span of time.

You'll notice the following lines at the top of the program:

```
import random,sys
from transposition import TransCipher
```

The `random` module will help us create random messages and keys to test. The `sys` (system) module is used for the command `sys.exit()`, so if a test fails we can quit the program. The second line ensures that we can use our `TransCipher` class as if it was embedded in this program. No extra syntax needed!

Optional challenge: import the module `time` and use it to time how long the tests take to run.

Reading and writing files

Up until now, we've been writing the messages we want to encrypt or decrypt directly into the Python program. What if we want to deal with longer messages? We could still copy and paste hundreds of lines directly into the program and print to the terminal, but it quickly becomes harder to deal with. Instead, we can use .txt files in the same directory as our python program.

Open the file filecipher.py. Parts of the code are already filled out for you and explained below. You need to complete the code under the TODO comments.
Use the code to encrypt Virginia Woolf's "A Room of One's Own" speech, then decrypt it and check that the code is working. Feel free to test your code with different text files and keys, just make sure not to overwrite a file you don't mean to!

```
input_file = 'woolf.txt'
output_file = 'woolf_encrypted.txt'
key = 24
mode = 'Encrypt' # or 'Decrypt'
```

Here we set the file we are translating and what we want to name the translated file. Keep in mind that whatever you set as `output_file` will get rewritten with the encrypted/decrypted text, so be careful with overwriting files. As we've seen before, `key` is an integer and `mode` sets which process we are doing.

```
if not os.path.exists(input_file):
    print(f"The file {input_file} does not exist")
    sys.exit()
```

Before opening the file, we check it actually exists. Python expects to find it in the same directory as the .py file. If it can't find it, we end the program using the command `sys.exit()`.

```
file_object = open(input_file)
content = file_object.read()
file_object.close()
```

Here we actually read the `input_file` and set a string with all the text to the variable `content`.

Next, you will need to write the code to time the process, do the actual encryption/decryption using the class `TransCipher`, and print how long the process took.

Hint: Use the method `time.time()` to measure the starting and ending times in seconds, then find the difference between them.

```
output_object = open(output_file, 'w') # write mode
output_object.write(translated)
output_object.close()
```

The last thing this program does is open (or create, if it doesn't exist yet) the `output_file` in 'write' mode, letting Python edit the file. It will copy the `translated` string into the file, overwriting whatever is already there, and close the file!

Test your file a few times with the `woolf.txt` file and another that you create.

Stop and think: How is having the ability to read and write files useful for an encryption program? What would the alternative be for long messages?

Detecting English

When we used brute force to hack the Caesar cipher, we had Python print out every possible decryption so we could read through them and recognize which one was correct. This was easy for us to do because humans are really good at pattern recognition - we can easily discern English from gibberish in a glance.

However, the transposition cipher has a lot more possible keys, which means more possible decryptions to look through. If we wanted to brute force the decryption of the Virginia Woolf speech, we would have to look at nearly 105,000 files! We need a better solution.

In order to lessen the amount of possible decryptions we need to look through, we are going to code a program that has Python guess if a string is in English or not. While it won't be perfect, it can narrow down the possibilities a lot!

Open the file `detectenglish.py`. The first function is completed for you, and your task is to complete the remaining three functions and then test your code.
Make sure the file `dictionary.txt` is in the same directory. This is a file with around 45000 english words, one on each line and all uppercase.

The function `create_dict(filename)` creates a dictionary whose keys are strings (all the words in the file `dictionary.txt`) and values are all `None`. Why would we create a dictionary with empty values if we could use a list to accomplish the same thing? The reason is that we will need to search through the dictionary many times to see if a word is in it, and searching through dictionaries is actually much faster than searching through lists. This has to do with the way dictionaries are implemented using hash tables, which is a very interesting but advanced topic. Here is an article about the topic if you are interested.

Follow the comments to implement the functions `remove_symbols`, `count_words`, and `isitenglish`. Then write some English, gibberish, and mixed phrases to test your code. Mess around with the values of `wordpercent` and `letterpercent` to understand how they work.

Hint: Use the method `.split()` to split your method into sets of characters separated by spaces, aka 'words'.

Stop and think: Why don't we have `wordpercent` set at 100 percent?

5 Hacking the transposition cipher

The last task is deceptively simple. We have already done the brunt of the work, and now we just need to fit the pieces together. We will input a message encrypted with a transposition cipher and Python will test every possible key to decrypt it. When it finds a possible decryption (determined by using `isitenglish()` from last section), Python will print for the user and ask if this is the decryption they are looking for. This way, the user does not have to look through thousands of gibberish decryptions, but our detect english program does not have to be perfect.

Open the file `hacker.py` and complete the function `hack(text)`. Add print statements such as `Hacking...` and `Attempting key #823` to make your code more entertaining!

As always, don't forget to test your code. When you believe your function is working well, decrypt the two secret messages at the top of the program.

Congratulations! You have successfully created a transposition encrypter, decrypter, and hacker. Reflect on all that you have learned and created in the past week, and be proud of yourself!

Below are some optional challenge ideas if you would like to explore further, or do your own research on cryptography!

Optional challenge: Use the file reading and writing techniques from `filecipher.py` and the hacking process from `hacker.py` to hack the file `mysteryscript.txt`.

Optional challenge: Explore the pyperclip module to be able to copy and paste from python easily. Instead of printing to the terminal, you could copy a decoded message to your clipboard to be able to paste is somewhere else.

Optional challenge: Learn a new cipher, such as the affine or multiplicative cipher. Swiegart's chapter on these ciphers is it great place to start, and includes an introduction to modular arithmetic.