Section 3

Data Structures
Errors & Exceptions
File Input/Output

Data Structures

A **data structure** is a way of organizing data in a computer so that it can be used effectively. The **type** of data structure you use depends on how you want to store information and how you plan to use it. Certain data structures are better suited for given tasks.

Lists

A **list** is a collection of items enclosed in **square brackets**. Items in a list are separated by **commas**. A list can have any number of items and they can be of different types.

You can use the **print()** function on a list and it will print the entire list as seen below.

Examples:

```
fruits = ["apple", "banana", "orange", "pear"]
nums = [1, 8, 3, 7, 11]
```

Accessing Elements in a List

Just as we indexed strings using square brackets, we can also use **indexing** to access individual elements in a list. An **index** is the location of an item in a list.

```
def main():      0   1   2   3
    myNums = [5, 7, 2, 11]

    print(myNums[1])  ⟶  7
```

Accessing Elements in a List

We can also use **slicing** to get a range of elements in a list.

Example:

```
drinks = ["water", "soda", "tea", "coffee", "orange juice", "smoothie"]
print(drinks[1:5])
```

```
['soda', 'tea', 'coffee', 'orange juice']
```

Manipulating Lists

Unlike strings which are **immutable** (cannot be modified in place), lists are **mutable**. We can edit an item in a list by referring to its index.

Example:

```
myList = ["one", "two", "three"]
myList[2] = "four"
print(myList)
```

```
['one', 'two', 'four']
```

Manipulating Lists

Be careful: since lists are **mutable**, changes to lists within a function **will be reflected** outside of the function.

```
def some_function(L):
    L[1] = "purple"

def main():

    myList = ["red", "blue", "yellow", "green"]
    some_function(myList)
    print(myList)
```

```
['red', 'purple', 'yellow', 'green']
```

Manipulating Lists

If you want to manipulate the list without changing the original one, you will need to **make a copy**. We can do this using the **.copy()** method.

```
def some_function(L):
    newList = L.copy()
    newList[1] = "purple"
    return newList

def main():

    myList = ["red", "blue", "yellow", "green"]
    anotherList = some_function(myList)
    print(myList)
    print(anotherList)
```

```
['red', 'blue', 'yellow', 'green']
['red', 'purple', 'yellow', 'green']
```

What will happen when you run this code?

```
colleges = ["Bryn Mawr", "Colgate", "Cornell", "Princeton", "Columbia"]
print("Harvard" in colleges)
```

What will happen when you run this code?

```
colleges = ["Bryn Mawr", "Colgate", "Cornell", "Princeton", "Columbia"]
print("Harvard" in colleges)
```

```
False
```

Iterating through a List

Similar to how we used a **for loop** to iterate through characters in a String, we can also use a for loop to iterate through **elements in a list** using the same syntax:

```
def main():
    myNums = [5, 7, 2, 11]

    for num in myNums:
        print(num)
```

```
5
7
2
11
```

Iterating through a List

We can also opt to loop by **index**, depending on the situation.

```
veggies = ["carrots", "broccoli", "spinach", "tomatoes"]

for i in range(len(veggies)):
    print(veggies[i])
```

```
carrots
broccoli
spinach
tomatoes
```

What will happen when you run this code?

```
nums = [4, 5, 3, 7, 5, 1]
count = 0

for num in nums:
    count += num

print(count)
```

What will happen when you run this code?

```
nums = [4, 5, 3, 7, 5, 1]
count = 0

for num in nums:
    count += num

print(count)
```

25

Adding Items to a List

We add items to a list using the method **.append()**. Append will add the given item to the **end of the list**.

```
odd = [1, 3, 5]

odd.append(7)
print(odd)
```

```
[1, 3, 5, 7]
```

Adding Items to a List

To insert an item at a specific position in a list, we use the method **.insert()**, which takes two parameters, the **index** you want to insert at and the **element** to insert.

```
myList = ['p', 'a', 'c', 'e']
myList.insert(1, 'e')
print(myList)
```

```
['p', 'e', 'a', 'c', 'e']
```

What will happen when you run this code?

```
def remove_empty(L):
    newList = []
    for item in L:
        if len(item) > 0:
            newList.append(item)
    return newList

def main():

    langs = ["English", "Spanish", "", "", "French", ""]
    print(remove_empty(langs))

if __name__ == "__main__":
    main()
```

What will happen when you run this code?

```
def remove_empty(L):
    newList = []
    for item in L:
        if len(item) > 0:
            newList.append(item)
    return newList

def main():

    langs = ["English", "Spanish", "", "", "French", ""]
    print(remove_empty(langs))

if __name__ == "__main__":
    main()
```

```
['English', 'Spanish', 'French']
```

Removing Items from a List

There are multiple ways to remove an item from a list. We can remove by **element** using **.remove()**, or we can remove by **index** using the keyword **del**.

Example:

```
holidays = ["Christmas", "Easter", "Halloween", "Thanksgiving"]

holidays.remove("Halloween")
del holidays[2]

print(holidays)
```

```
['Christmas', 'Easter', 'Thanksgiving']
```

Operations on Lists

We can use the "+" sign to concatenate lists, and the "*" sign to repeat a list a given number of times.

Example:

```
even = [2, 4, 6, 8]
even += [10, 12]
print(even)  ⟶  [2, 4, 6, 8, 10, 12]
print([2] * 4)  ⟶  [2, 2, 2, 2]
```

Common List Methods

.append(x) - adds an element x to the end of a list

.remove(x) - removes first item x from a list

.count(x) - returns the number of times x appears in the list

.index(x) - returns the index (first occurrence) of x, returns error if not in list

.sort() - sorts the list

len(list) - returns the length of the list

What will happen when you run this code?

```
def largest(L):

    largest = 0

    for num in L:
        if num > largest:
            largest = num

    return largest

def main():

    print(largest([20, 6, 7, 14, 21, 17]))

if __name__ == "__main__":
    main()
```

What will happen when you run this code?

```
def largest(L):

    largest = 0

    for num in L:
        if num > largest:
            largest = num

    return largest

def main():

    print(largest([20, 6, 7, 14, 21, 17]))

if __name__ == "__main__":
    main()
```

21

What will happen when you run this code?

```
cart = ["milk", "eggs", "detergent", "milk", "toilet paper"]

for item in cart:
    print(cart.count(item))
```

What will happen when you run this code?

```
cart = ["milk", "eggs", "detergent", "milk", "toilet paper"]

for item in cart:
    print(cart.count(item))
```

```
2
1
1
2
1
```

Dictionaries

A dictionary is a container that stores multiple items as **key:value** pairs. Values in dictionaries are referenced by their associated key. All keys in a dictionary **must be unique**. Dictionaries are defined using **curly braces {}** and elements are separated by **commas**.

```
student = {"name": "John", "age": 20, "birthday": "09/23/99", "grades": [97, 94, 95]}
```

Accessing Elements in a Dictionary

Dictionaries are **unordered**, which means we cannot use indexing on them. Instead, we can refer to an element's **key** name.

```
student = {"name": "John", "age": 20, "birthday": "09/23/99", "grades": [97, 94, 95]}
print(student["name"])
print(student["grades"])
```

```
John
[97, 94, 95]
```

Manipulating Elements in a Dictionary

To update a key in a dictionary, we can refer to the **key** we want to change and set it to a new **value**.

```
temps = {"red": "hot", "blue": "cold"}
temps["red"] = "warm"
print(temps)
```

```
{'red': 'warm', 'blue': 'cold'}
```

What will happen when you run this code?

```
student = {"name": "John", "age": 20, "birthday": "09/23/99", "grades": [97, 94, 95]}
student["grades"].append(88)
print(student["grades"])
```

What will happen when you run this code?

```
student = {"name": "John", "age": 20, "birthday": "09/23/99", "grades": [97, 94, 95]}
student["grades"].append(88)
print(student["grades"])
```

```
[97, 94, 95, 88]
```

What will happen when you run this code?

```
def count_fruits(basket):

    fruits = {}

    for fruit in basket:
        fruits[fruit] += 1

    return fruits

def main():
    print(count_fruits(["apple", "orange", "orange", "pear", "banana", "apple"]))

if __name__ == "__main__":
    main()
```

What will happen when you run this code?

```
def count_fruits(basket):

    fruits = {}

    for fruit in basket:
        fruits[fruit] += 1

    return fruits

def main():
    print(count_fruits(["apple", "orange", "orange", "pear", "banana", "apple"]))

if __name__ == "__main__":
    main()
```

```
Traceback (most recent call last):
  File "/Users/brittchin/Desktop/python workshop/testing.py", line 76, in <module>
    main()
  File "/Users/brittchin/Desktop/python workshop/testing.py", line 73, in main
    print(count_fruits(["apple", "orange", "orange", "pear", "banana", "apple"]))
  File "/Users/brittchin/Desktop/python workshop/testing.py", line 68, in count_fruits
    fruits[fruit] += 1
KeyError: 'apple'
```

How do we fix this?

```
def count_fruits(basket):

    fruits = {}

    for fruit in basket:
        if fruit in fruits:
            fruits[fruit] += 1
        else:
            fruits[fruit] = 1

    return fruits

def main():
    print(count_fruits(["apple", "orange", "orange", "pear", "banana", "apple"]))

if __name__ == "__main__":
    main()
```

```
{'apple': 2, 'orange': 2, 'pear': 1, 'banana': 1}
```

Iterating through a Dictionary

We can use **for loops** to loop through elements in a dictionary. We can do this multiple ways. To loop through the **keys** in a dictionary, we can do the following:

```
cost = {"orange": 2.50, "banana": 1.10, "apple": 2.00}

for key in cost:
    print(key)
```

```
orange
banana
apple
```

Iterating through a Dictionary

To loop through the **values** in a dictionary, we can do the following:

```
cost = {"orange": 2.50, "banana": 1.10, "apple": 2.00}

for key in cost:
    print(cost[key])
```

```
2.5
1.1
2.0
```

Iterating through a Dictionary

To loop through all **key, value pairs** in a dictionary, we use the **.items()** method:

```
cost = {"orange": 2.50, "banana": 1.10, "apple": 2.00}

for key, value in cost.items():
    print("{} costs ${}0.".format(key, value))
```

```
orange costs $2.50.
banana costs $1.10.
apple costs $2.00.
```

What will happen when you run this code?

```
students = [{"name": "Mary", "rank": 3},
            {"name": "Jose", "rank": 2},
            {"name": "Anna", "rank": 1}]

for student in students:
    print(student["name"])
```

What will happen when you run this code?

```
students = [{"name": "Mary", "rank": 3},
            {"name": "Jose", "rank": 2},
            {"name": "Anna", "rank": 1}]

for student in students:
    print(student["name"])
```

```
Mary
Jose
Anna
```

What will happen when you run this code?

```
tours = [{
    "name": "Backpacking in Sydney",
    "guide": "Sally",
    "tags": ["nature", "outdoors", "hiking"]},
    {
    "name": "Venice Food Tour",
    "guide": "Jacob",
    "tags": ["food", "outdoors", "kid-friendly"]},
    {
    "name": "Adirondack Hike",
    "guide": "Peter",
    "tags": ["day-trip", "nature"]}]

for tour in tours:
    if "nature" in tour["tags"]:
        print(tour["name"])
```

What will happen when you run this code?

```
tours = [{
    "name": "Backpacking in Sydney",
    "guide": "Sally",
    "tags": ["nature", "outdoors", "hiking"]},
    {
    "name": "Venice Food Tour",
    "guide": "Jacob",
    "tags": ["food", "outdoors", "kid-friendly"]},
    {
    "name": "Adirondack Hike",
    "guide": "Peter",
    "tags": ["day-trip", "nature"]}]

for tour in tours:
    if "nature" in tour["tags"]:
        print(tour["name"])
```

```
Backpacking in Sydney
Adirondack Hike
```

Adding Items to a Dictionary

To add an item to a dictionary, we create a new **key** and assign a **value** to it.

```
squares = {1: 1, 3: 9, 5: 25, 7: 49}
squares[9] = 81

print(squares)
```

```
{1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
```

Removing Items from a Dictionary

To remove an item from a dictionary, we can use the method **.pop()** or the keyword **del**.

```
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
squares.pop(3)

print(squares)
```

```
{1: 1, 5: 25, 7: 49, 9: 81}
```

Data Structures Practice

In the starter code folder, open the file **data_structures_practice.py** in the **section 3** folder. Work on filling out the three functions in that file.

in_list() - takes a list L and an int x and returns True if the element x appears in L and False if it does not

most_occurances() - takes a list L and returns the element that appears the most times

highest_average() - takes a dictionary where the keys are students and the values are lists of grades, and returns the student with the highest average (sum of grades/num of grades)

File Input/Output (I/O)

Handling files is an important part of any application. **Files** are used to permanently store data in memory (ex: on your local computer)

Opening Files

Before a file can be read or written, it must be **opened**. We open a file using the **open()** function, which takes two parameters: the **file name** and **open mode**.

r - **Read**: the default mode, opens a file for reading, error if file does not exist

a - **Append**: opens a file for appending, creates the file if it does not exist

w - **Write**: opens a file for writing, creates the file if it does not exist

x - **Create**: creates the specified file, returns an error if the file exists

Opening Files

To open a file for reading, use the following syntax:

```
f = open("test.txt")
```

The file **must exist** in order to open it. In this example, “test.txt” would have to be in the same folder as your python file to be read. Since no mode is provided, the file will automatically be open in “read” mode.

Reading Files

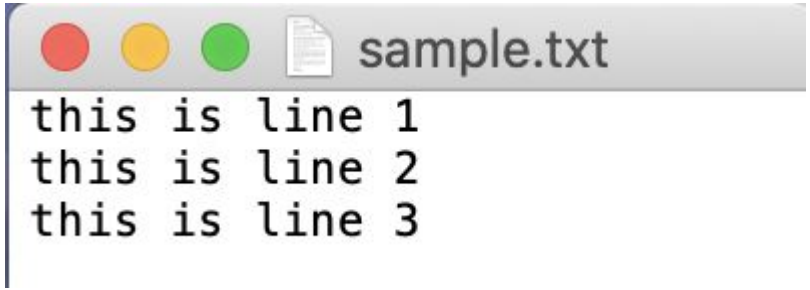
After a file has been opened, it can be **read**. By default, **.read()** will return the entire file. Think of read like a cursor scrolling through the entire document.

```
file = open("sample.txt", 'r')

print(file.read())
```

```
this is line 1
this is line 2
this is line 3
```

Reading Files

You can also read a single line in the file by using the method **.readline()**.

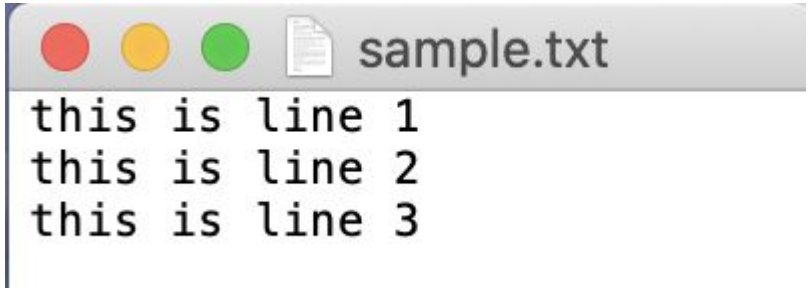



```
file = open("sample.txt", 'r')

print(file.readline())
print(file.readline())
print(file.readline())
```

```
this is line 1

this is line 2

this is line 3
```

Reading Files

To read an entire file **line by line**, we can simply use a **for loop**.

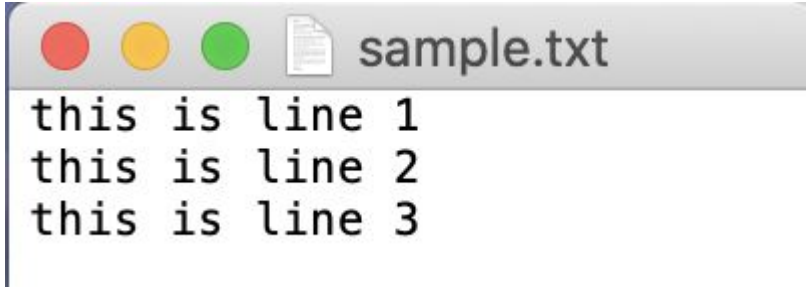



```
f = open("sample.txt", 'r')

for line in f:
    print(line)
```

```
this is line 1

this is line 2

this is line 3
```

Reading Files

As you can see, there are **blank lines** between each line in the file. This is because each line in the file has an invisible **newline ('\n') character** at the end of it. When we call print(), it automatically adds another newline. To fix this, we can add **end=''**.




```
file = open("sample.txt", 'r')

for line in file:
    print(line, end='')

this is line 1
this is line 2
this is line 3
```

Writing Files

All of the information from your program **disappears once it stops running**. Therefore, we need to save information outside of the program file so we don't lose it. We do this by **writing to a file**. We can use one of two modes: **"w"** or **"a"**. Using "w" will **overwrite all existing data in the file**, whereas "a" will **append to the end of the file**.

Writing Files

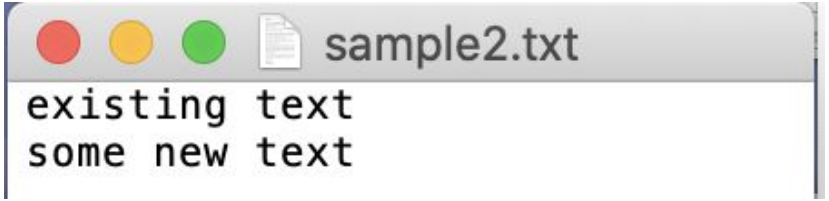
Example:


```
file = open("sample2.txt", 'w')
file.write("some new text")
```

Writing Files

Example:


```
file = open("sample2.txt", 'a')
file.write("\nsome new text")
```



Writing Files

You can also **create a new file** using write.

```
file = open("newFile.txt", 'w')
file.write("this is a new file")
```

Closing Files

It is good practice to close your file once you are done with it. We do this by calling **.close()**.

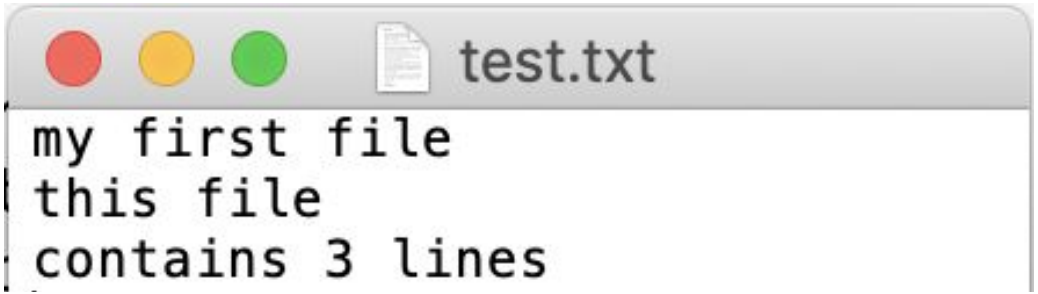
```
file = open("sample.txt", 'r')

for line in file:
    print(line, end='')

file.close()
```

What will happen when you run this code?

```
f = open("test.txt", "w")
f.write("my first file\n")
f.write("this file\n")
f.write("contains 3 lines\n")
f.close()
```

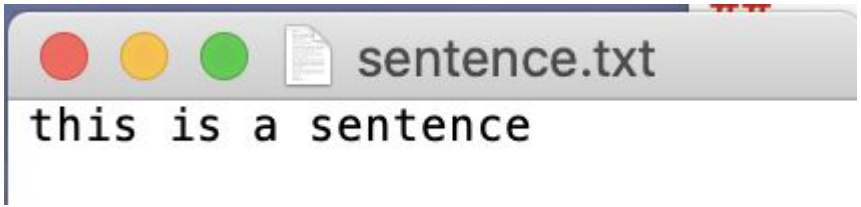
What will happen when you run this code?

```
f = open("test.txt", "w")
f.write("my first file\n")
f.write("this file\n")
f.write("contains 3 lines\n")
f.close()
```



Getting the length of a file

We can use the **len()** function to get the **number of characters** in a file.




```
f = open("sentence.txt", 'r')

length = len(f.read())
print(length)
```

```
18
```

Seeking

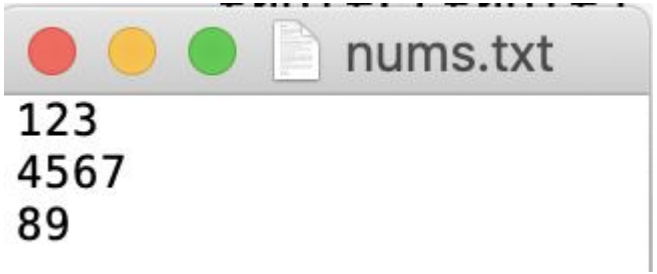
Since calling **.read()** moves the cursor to the bottom of the file, we can use **.seek()** to go to a specific index in the document. **filename.seek(0)** returns the cursor to the beginning of the file.

```
file = open("sample.txt", 'r')
length = len(file.read())
file.seek(0)

for line in file:
    print(line, end='')
```

```
this is line 1
this is line 2
this is line 3
```

What will happen when you run this code?

```
file = open("nums.txt", 'r')
length = len(file.read())
word = "cat"
times = (length // len(word)) * word
file.seek(0)
i = 0

for line in file:
    for char in line:
        if char.isdigit():
            print(times[i] + char)
            i += 1
```

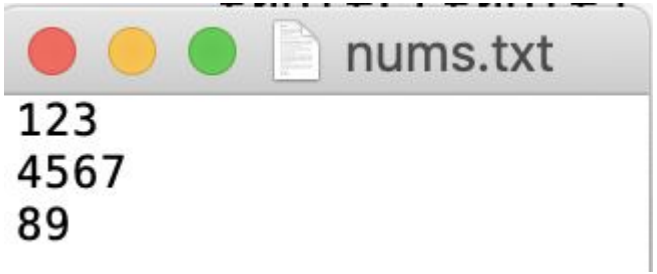


What will happen when you run this code?

```
file = open("nums.txt", 'r')
length = len(file.read())
word = "cat"
times = (length // len(word)) * word
file.seek(0)
i = 0

for line in file:
    for char in line:
        if char.isdigit():
            print(times[i] + char)
            i += 1
```

11

catcatcat



c1
a2
t3
c4
a5
t6
c7
a8
t9

Errors

By now you have probably experienced many different errors in Python. There are many types of errors, including but not limited to:

Syntax Error - error results before the program runs due to code that does not conform to the language rules

Runtime Error - error results after the program has begun running

Semantic Error - the program does not throw an error, but the output is not what it should be

Errors

Python has many exceptions that are raised when your program encounters an error (something in the program goes wrong). If you do not handle these errors, **your program will crash**. You should always consider the possible runtime errors and write code to prevent this from happening.

Error Handling

Debugging is one way of handling errors. However, errors can also be handled within your code to ensure that your program does not crash.

Examples:

NameError - object with the given name cannot be found

IndexError - the index you are trying to access is not valid

KeyError - a key you are trying to access is not in the dictionary

Error Handling

```
animals = {"dog": "woof", "cat": "meow", "horse": "neigh"}
print(animals["cow"])
```

```
Traceback (most recent call last):
  File "/Users/brittchin/Desktop/python workshop/testing.py", line 88, in <module>
    main()
  File "/Users/brittchin/Desktop/python workshop/testing.py", line 85, in main
    print(animals["cow"])
KeyError: 'cow'
```

Error Handling

```
myFile = open("filedoesntexist.txt")
```

```
Traceback (most recent call last):
  File "/Users/brittchin/Desktop/python workshop/testing.py", line 91, in <module>
    main()
  File "/Users/brittchin/Desktop/python workshop/testing.py", line 82, in main
    myFile = open("filedoesntexist.txt")
FileNotFoundError: [Errno 2] No such file or directory: 'filedoesntexist.txt'
```

Try/Catch

A **try/catch** is used to handle exception errors. You can specify a specific error, or use **except** to catch all errors.

```
try:
    something
except: #an error occurs
    handle it
```

Try/Catch

Example:

```
try:
    myFile = open("filedoesntexist.txt")
except FileNotFoundError:
    print("File entered was not found.")
```

```
File entered was not found.
```

Section 3 Summary

- A **list** is an array that can store multiple items of data in sequentially numbered elements that start at zero
- A **dictionary** stores key: value pairs and are unordered
- Files must be opened before they can be read or written to
- Anticipated runtime exception errors can be handled with a **try/except** block

SECTION 3 PROJECT

Message Encryptor

Message Encryptor

You are working as a secret agent at a top secret government firm. For your current mission, you have been assigned a partner to correspond with over your super secure email server. However, you do not want your message to be read if hackers happen to intercept it, as it contains extremely sensitive information. You must find a way to encrypt your messages so only those with the secret key will be able to read them.

Message Encryptor

You have been provided with starter code (**message_encryptor_starter.py**) and instructions (**message_encryptor.PDF**) for a message encryptor system. You will need to complete the given functions to encrypt and decrypt the given files.