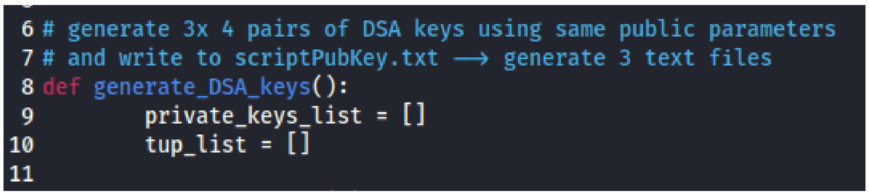
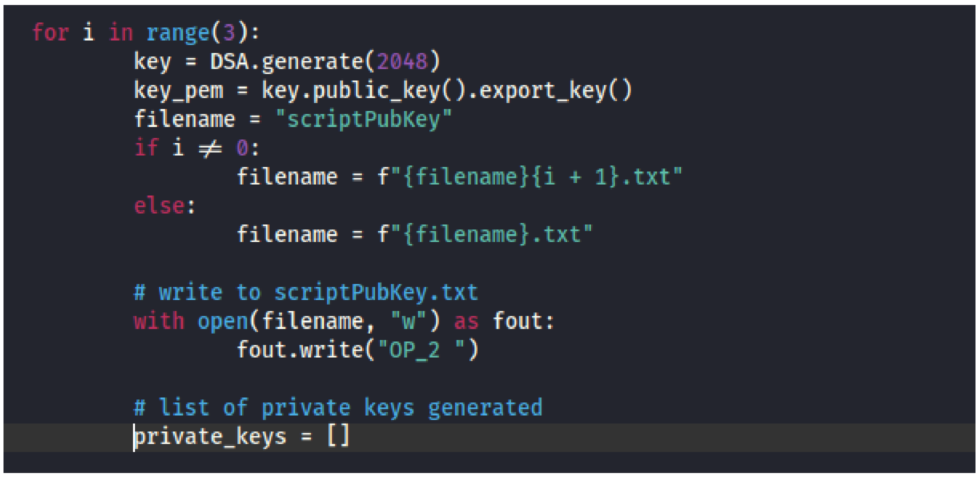
*Python version 3.10.5

<u>Generate DSA keys:</u>



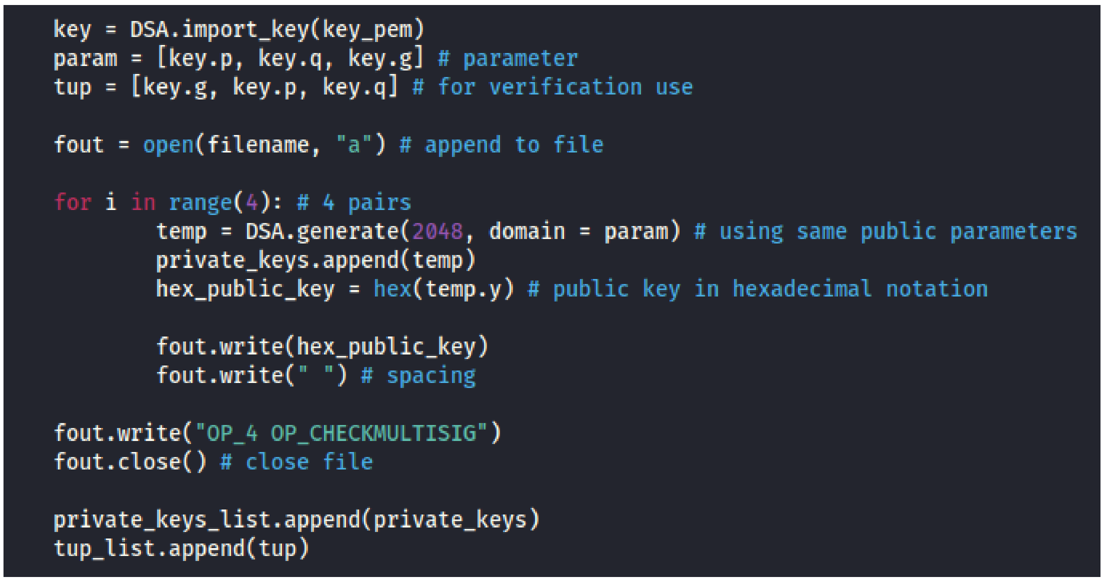
```
# generate 3x 4 pairs of DSA keys using same public parameters
# and write to scriptPubKey.txt ⟶ generate 3 text files
def generate_DSA_keys():
        private_keys_list = []
        tup_list = []

```

- Firstly, what I did was I created a two empty lists, one to store the list of private keys and the other to store the list to tups as it is required for me to generate 3 instances of scriptPubKey



```
for i in range(3):
        key = DSA.generate(2048)
        key_pem = key.public_key().export_key()
        filename = "scriptPubKey"
        if i ≠ 0:
                filename = f"{filename}{i + 1}.txt"
        else:
                filename = f"{filename}.txt"

        # write to scriptPubKey.txt
        with open(filename, "w") as fout:
                fout.write("OP_2 ")

        # list of private keys generated
        private_keys = []
```

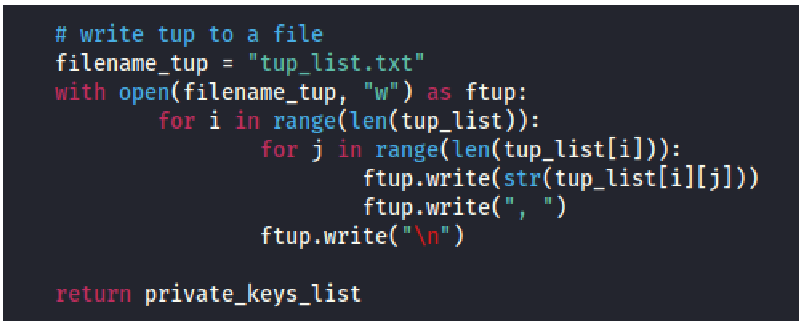
- Within the same function, I performed a loop to create generate 3 scriptPubKey text files and with the first text file with the default name ‘scriptPubKey.txt’ and the subsequent two files as ‘scriptPubKey2.txt’ and ‘scriptPubKey3.txt’.



```
key = DSA.import_key(key_pem)
param = [key.p, key.q, key.g] # parameter
tup = [key.g, key.p, key.q] # for verification use

fout = open(filename, "a") # append to file

for i in range(4): # 4 pairs
        temp = DSA.generate(2048, domain = param) # using same public parameters
        private_keys.append(temp)
        hex_public_key = hex(temp.y) # public key in hexadecimal notation

        fout.write(hex_public_key)
        fout.write(" ") # spacing

fout.write("OP_4 OP_CHECKMULTISIG")
fout.close() # close file

private_keys_list.append(private_keys)
tup_list.append(tup)
```

- The program then continues to generate the public key and private keys using the same public parameters for each iteration of the loop and add the private keys to a list and converts the public key to a hexadecimal notation.

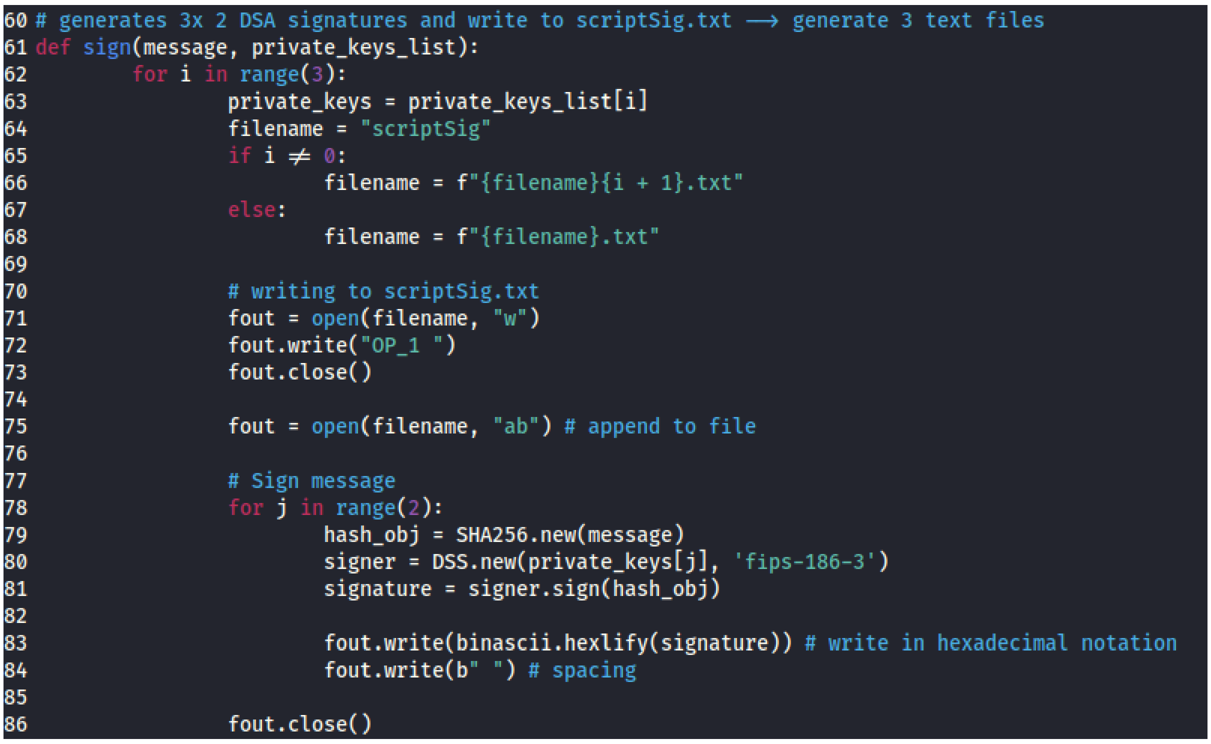
- The converted public key is then written to the file while the set of private keys are added to the variable ‘private_keys_list’, which stores the all the sets of private keys generated at each iteration of the loop.
- The tup that was initialised previously is also added to the variable ‘tup_list’, which stores all the sets of verification variables generated by each iteration of the loop.



```python
    # write tup to a file
    filename_tup = "tup_list.txt"
    with open(filename_tup, "w") as ftup:
            for i in range(len(tup_list)):
                    for j in range(len(tup_list[i])):
                            ftup.write(str(tup_list[i][j]))
                            ftup.write(", ")
                    ftup.write("\n")

    return private_keys_list
```

- The variables required for the verification process is then written to a text file which can then be read from the signature verification program.
- Finally, the function then returns the list of private keys which will be used to sign the message.

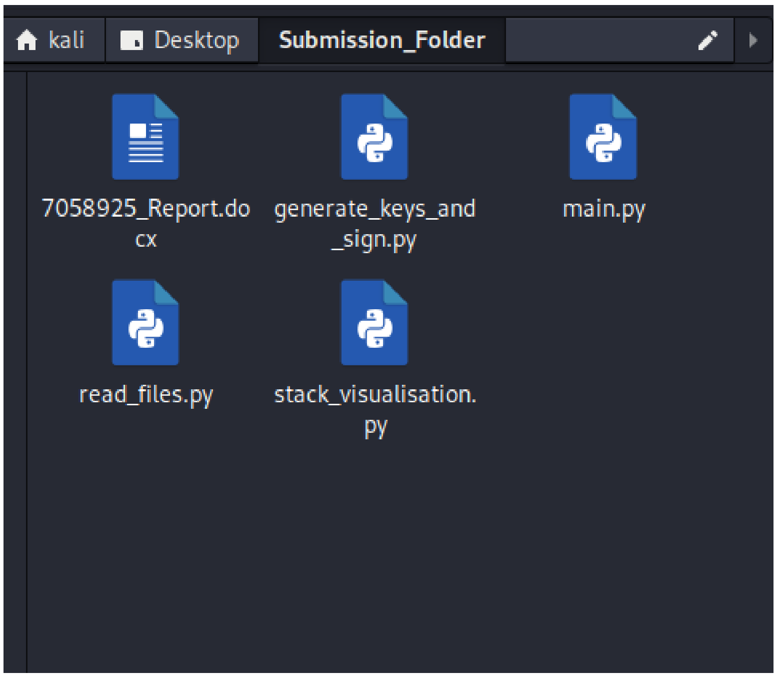
<u>Signing the message:</u>



```python
# generates 3x 2 DSA signatures and write to scriptSig.txt ⟶ generate 3 text files
def sign(message, private_keys_list):
        for i in range(3):
                private_keys = private_keys_list[i]
                filename = "scriptSig"
                if i ≠ 0:
                        filename = f"{filename}{i + 1}.txt"
                else:
                        filename = f"{filename}.txt"

                # writing to scriptSig.txt
                fout = open(filename, "w")
                fout.write("OP_1 ")
                fout.close()

                fout = open(filename, "ab") # append to file

                # Sign message
                for j in range(2):
                        hash_obj = SHA256.new(message)
                        signer = DSS.new(private_keys[j], 'fips-186-3')
                        signature = signer.sign(hash_obj)

                        fout.write(binascii.hexlify(signature)) # write in hexadecimal notation
                        fout.write(b" ") # spacing

                fout.close()
```

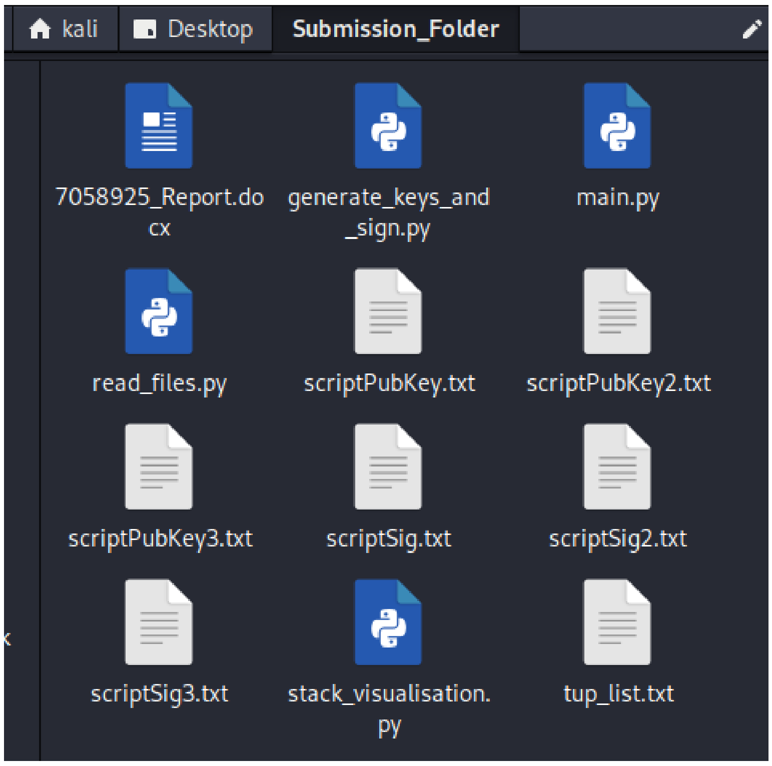
- Same with the DSA keys, it is required for me to generate 3 scriptSig files. Hence, a loop is created to do just that.
- By using the return value(private_keys_list) generated from the previous function, each iteration of the loop will extract the appropriate list of private keys that is required to be used for the signing of the message.
- The result of the signing process, the signature, is then converted to a hexadecimal notation and then written to the text file with the same naming pattern as scriptPubKey i.e., ‘scriptSig.txt’, ‘scriptSig2.txt’, ‘scriptSig3.txt’.

Before running the above functions:



7058925_Report.docx
generate_keys_and_sign.py
main.py
read_files.py
stack_visualisation.py

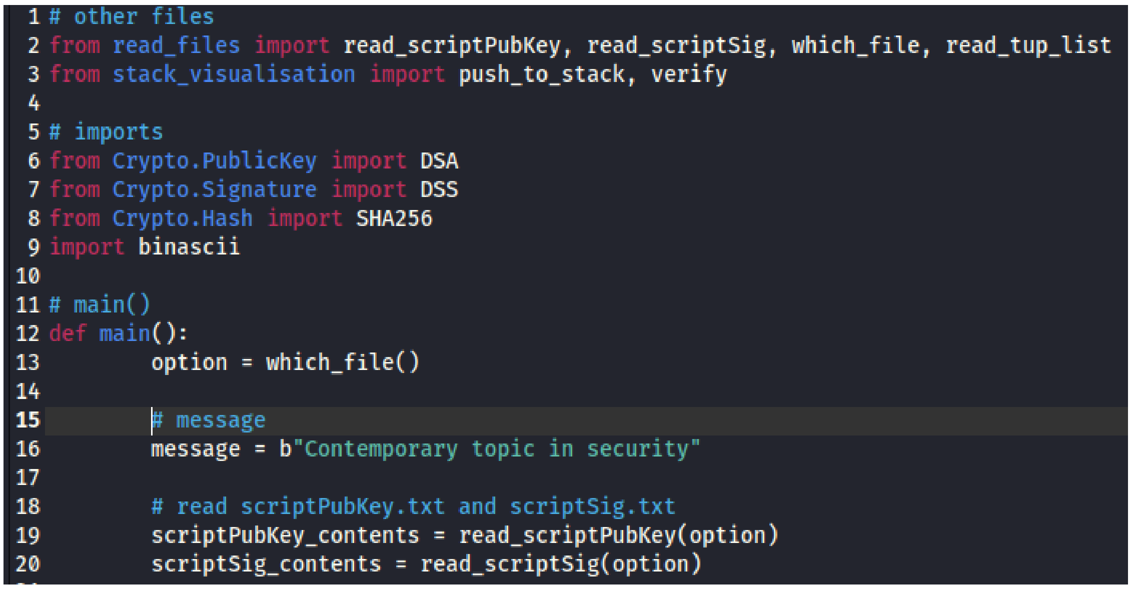
After running the above functions:



7058925_Report.docx
generate_keys_and_sign.py
main.py
read_files.py
scriptPubKey.txt
scriptPubKey2.txt
scriptPubKey3.txt
scriptSig.txt
scriptSig2.txt
scriptSig3.txt
stack_visualisation.py
tup_list.txt

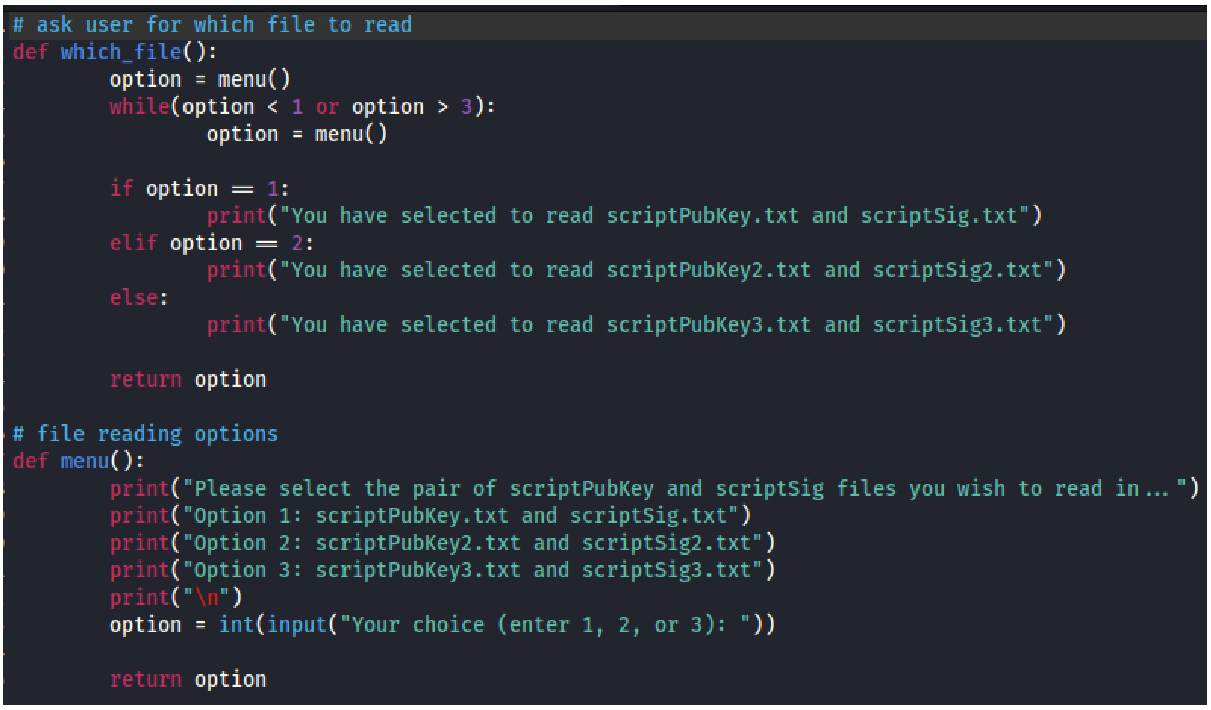
- 3 scriptPubKey text files and 3 scriptSig text files are generated
- An additional text file named ‘tup_list.txt’ is generated for the signature verification process

Driver file – main.py:



```python
# other files
from read_files import read_scriptPubKey, read_scriptSig, which_file, read_tup_list
from stack_visualisation import push_to_stack, verify

# imports
from Crypto.PublicKey import DSA
from Crypto.Signature import DSS
from Crypto.Hash import SHA256
import binascii

# main()
def main():
        option = which_file()

        # message
        message = b"Contemporary topic in security"

        # read scriptPubKey.txt and scriptSig.txt
        scriptPubKey_contents = read_scriptPubKey(option)
        scriptSig_contents = read_scriptSig(option)
```

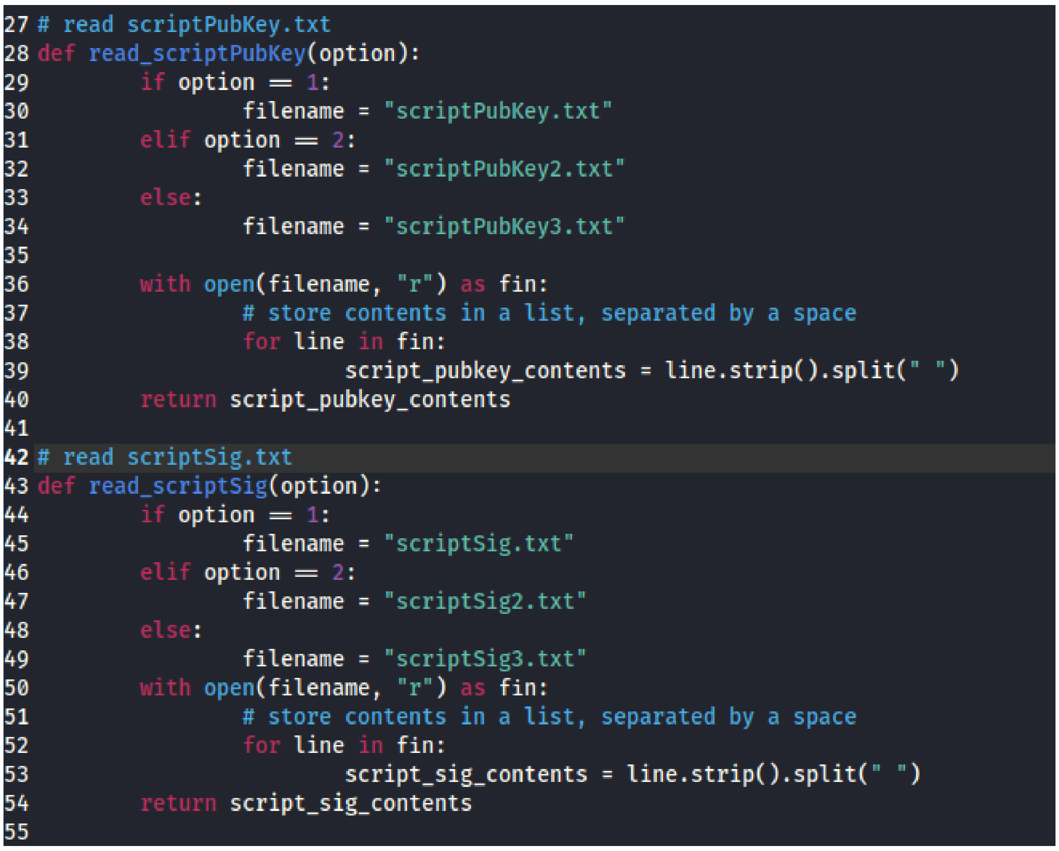
- Firstly, all the relevant imports are included at the top of the file. These imports include other python files which is required for the whole program to run.
- Because there are three pairs of scriptPubKey and scriptSig text files, the first thing the program does is to ask the user which file they want to read. The code is illustrated in the screenshot below.



```python
# ask user for which file to read
def which_file():
        option = menu()
        while(option < 1 or option > 3):
                option = menu()

        if option == 1:
                print("You have selected to read scriptPubKey.txt and scriptSig.txt")
        elif option == 2:
                print("You have selected to read scriptPubKey2.txt and scriptSig2.txt")
        else:
                print("You have selected to read scriptPubKey3.txt and scriptSig3.txt")

        return option

# file reading options
def menu():
        print("Please select the pair of scriptPubKey and scriptSig files you wish to read in...")
        print("Option 1: scriptPubKey.txt and scriptSig.txt")
        print("Option 2: scriptPubKey2.txt and scriptSig2.txt")
        print("Option 3: scriptPubKey3.txt and scriptSig3.txt")
        print("\n")
        option = int(input("Your choice (enter 1, 2, or 3): "))

        return option
```

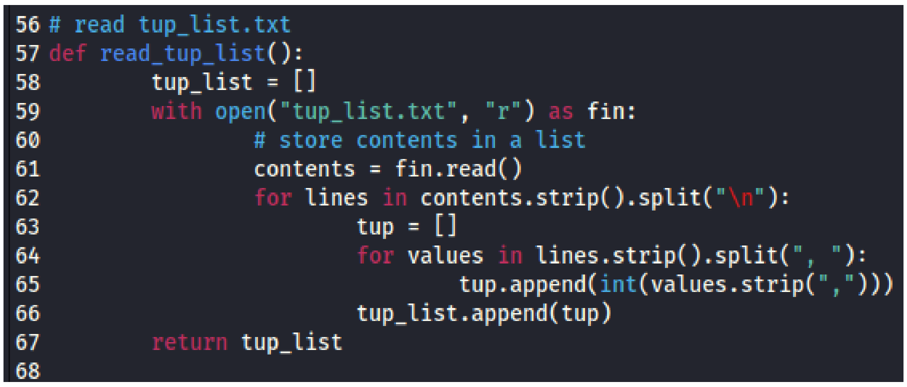
- The user will then enter '1', '2', or '3' depending on which files they want to read. The corresponding file will then be read by the program when the program.

Reading files:



```python
# read scriptPubKey.txt
def read_scriptPubKey(option):
        if option == 1:
                filename = "scriptPubKey.txt"
        elif option == 2:
                filename = "scriptPubKey2.txt"
        else:
                filename = "scriptPubKey3.txt"

        with open(filename, "r") as fin:
                # store contents in a list, separated by a space
                for line in fin:
                        script_pubkey_contents = line.strip().split(" ")
        return script_pubkey_contents

# read scriptSig.txt
def read_scriptSig(option):
        if option == 1:
                filename = "scriptSig.txt"
        elif option == 2:
                filename = "scriptSig2.txt"
        else:
                filename = "scriptSig3.txt"
        with open(filename, "r") as fin:
                # store contents in a list, separated by a space
                for line in fin:
                        script_sig_contents = line.strip().split(" ")
        return script_sig_contents

```

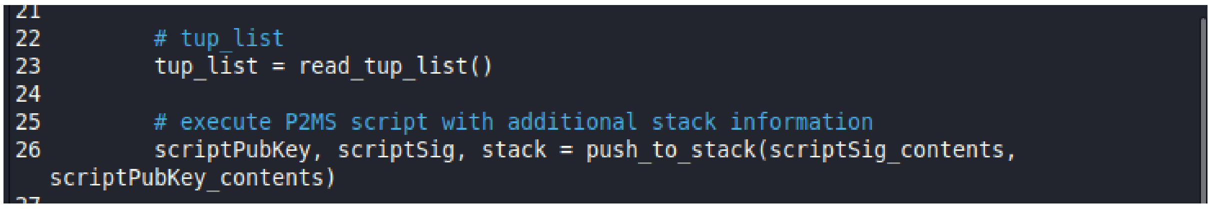
- Depending on the user's input, these two functions above will read the contents of the corresponding file and return the value of the file contents.



```python
# read tup_list.txt
def read_tup_list():
        tup_list = []
        with open("tup_list.txt", "r") as fin:
                # store contents in a list
                contents = fin.read()
                for lines in contents.strip().split("\n"):
                        tup = []
                        for values in lines.strip().split(", "):
                                tup.append(int(values.strip(",")))
                        tup_list.append(tup)
        return tup_list

```

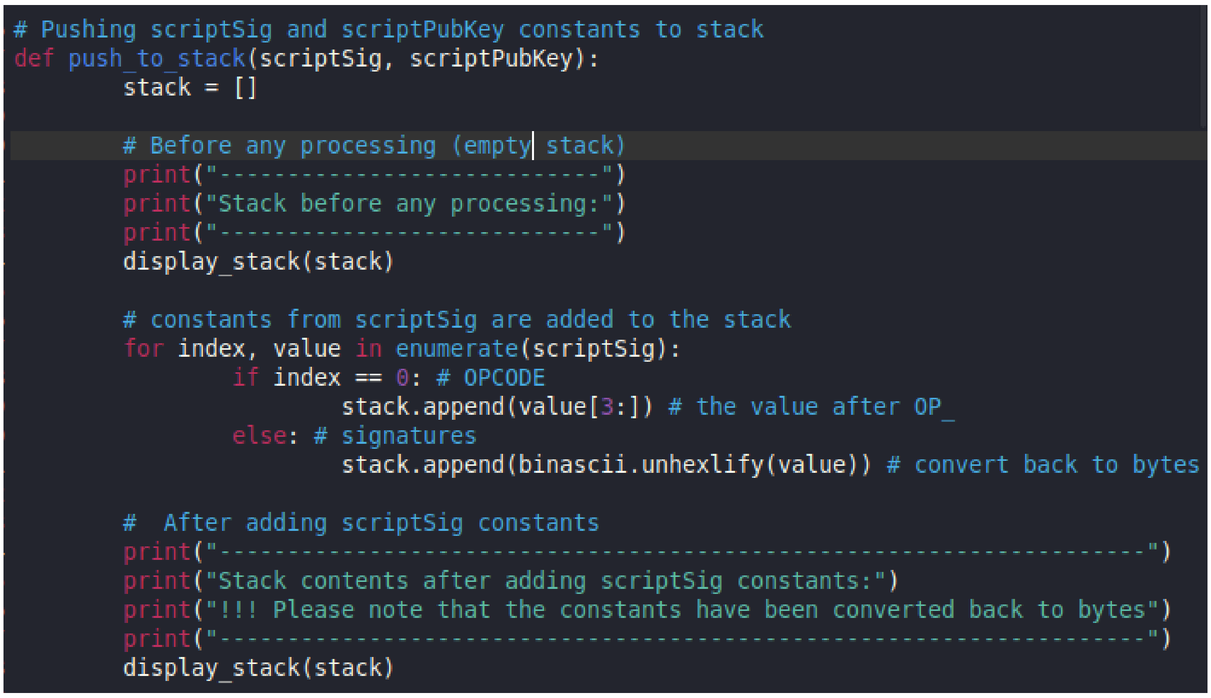
- The above function is to read 'tup_list.txt' file. The file contents are of <type: string>, but the original format of the content is stored as a 2-dimensional list. Therefore, the function converts and returns the contents to a 2D list as per its original format before it was written to the file.

Driver program – main.py (continued):



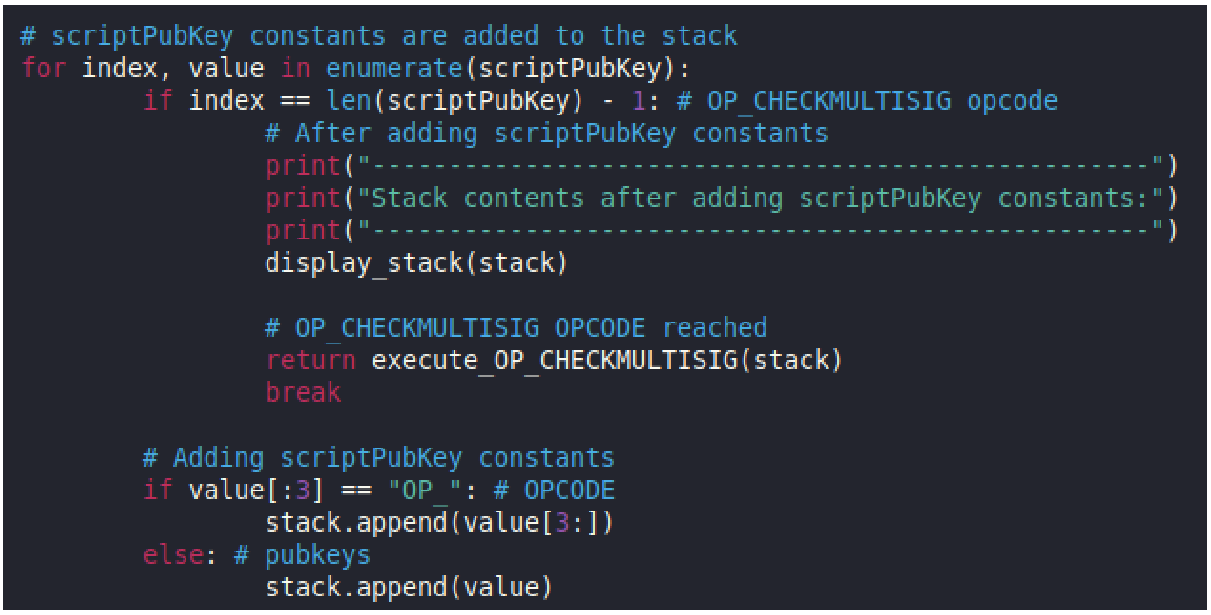
```
    # tup_list
    tup_list = read_tup_list()

    # execute P2MS script with additional stack information
    scriptPubKey, scriptSig, stack = push_to_stack(scriptSig_contents,
scriptPubKey_contents)
```

- After reading all the necessary files, the main program then executes a function which pushes the relevant contents to a stack.



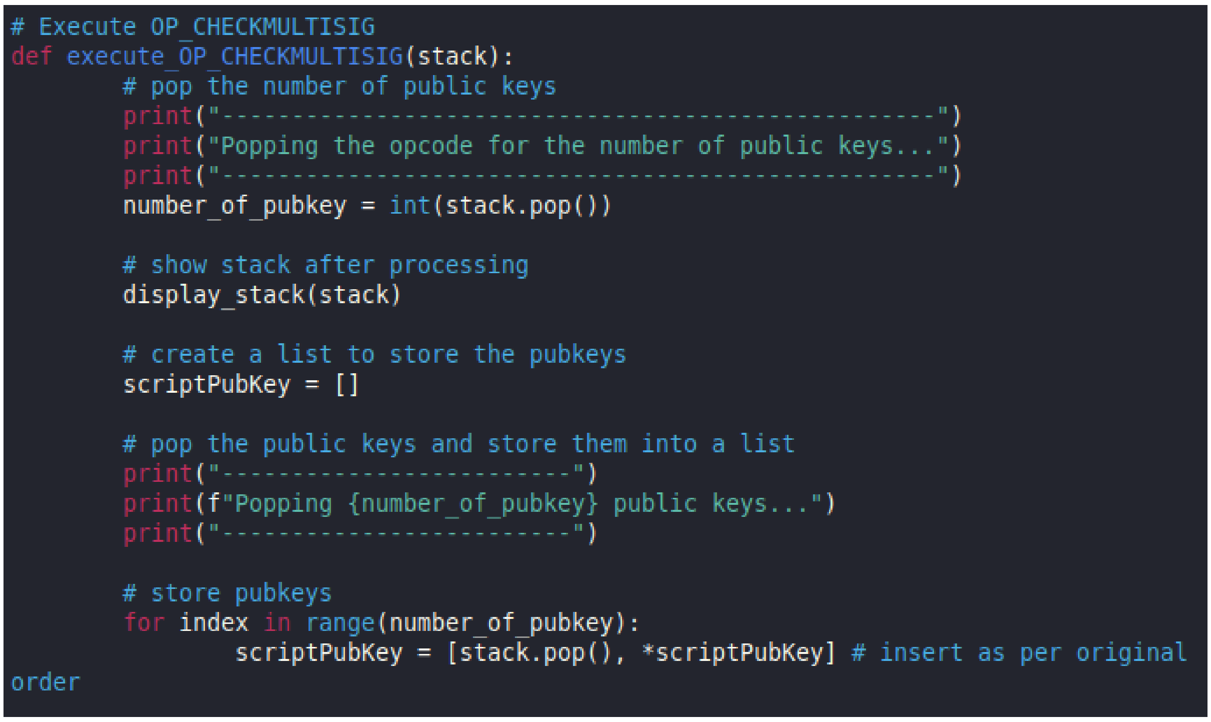
```
# Pushing scriptSig and scriptPubKey constants to stack
def push_to_stack(scriptSig, scriptPubKey):
        stack = []

        # Before any processing (empty stack)
        print("----------------------------")
        print("Stack before any processing:")
        print("----------------------------")
        display_stack(stack)

        # constants from scriptSig are added to the stack
        for index, value in enumerate(scriptSig):
                if index == 0: # OPCODE
                        stack.append(value[3:]) # the value after OP_
                else: # signatures
                        stack.append(binascii.unhexlify(value)) # convert back to bytes

        #  After adding scriptSig constants
        print("-------------------------------------------------------------------")
        print("Stack contents after adding scriptSig constants:")
        print("!!! Please note that the constants have been converted back to bytes")
        print("-------------------------------------------------------------------")
        display_stack(stack)
```

- When the above function is called, a list is created to represent a stack structure. The first thing this function does is to push the values of scriptSig constants into the stack. Note that before pushing the signatures into the stack, I converted them from hexadecimal notation to its original bytes format. Afterwards, the contents of the stack are printed out for visualisation.



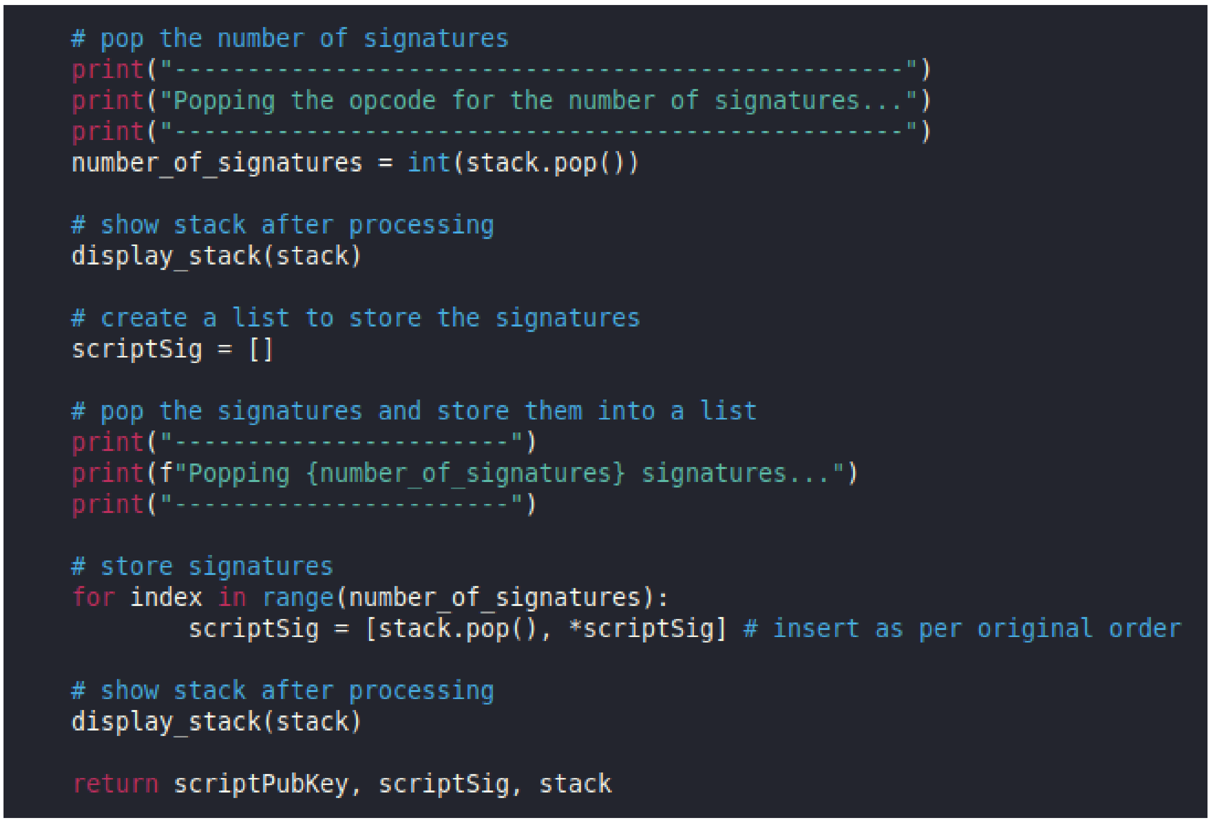
```python
# scriptPubKey constants are added to the stack
for index, value in enumerate(scriptPubKey):
        if index == len(scriptPubKey) - 1: # OP_CHECKMULTISIG opcode
                # After adding scriptPubKey constants
                print("---------------------------------------------------")
                print("Stack contents after adding scriptPubKey constants:")
                print("---------------------------------------------------")
                display_stack(stack)

                # OP_CHECKMULTISIG OPCODE reached
                return execute_OP_CHECKMULTISIG(stack)
                break

        # Adding scriptPubKey constants
        if value[:3] == "OP_": # OPCODE
                stack.append(value[3:])
        else: # pubkeys
                stack.append(value)
```

- After adding scriptSig constants, scriptPubKey constants are pushed next. Once the loop reaches the OP_CHECKMULTISIG opcode, it will call another function which processes the stack contents.



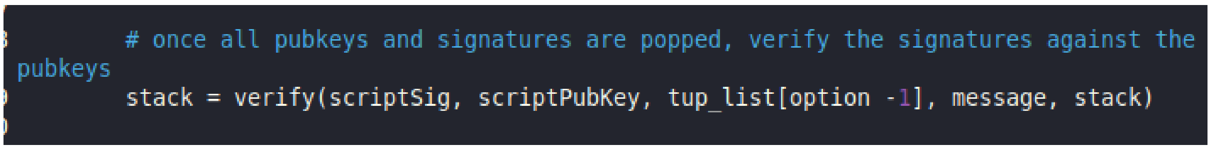
```python
# Execute OP_CHECKMULTISIG
def execute_OP_CHECKMULTISIG(stack):
        # pop the number of public keys
        print("---------------------------------------------------")
        print("Popping the opcode for the number of public keys...")
        print("---------------------------------------------------")
        number_of_pubkey = int(stack.pop())

        # show stack after processing
        display_stack(stack)

        # create a list to store the pubkeys
        scriptPubKey = []

        # pop the public keys and store them into a list
        print("-------------------------")
        print(f"Popping {number_of_pubkey} public keys...")
        print("-------------------------")

        # store pubkeys
        for index in range(number_of_pubkey):
                scriptPubKey = [stack.pop(), *scriptPubKey] # insert as per original
order
```

- The above function executes once the program captures the OP_CHECKMULTISIG opcode. This function pops out the stack contents as per a stack data structure – First-In-Last-Out (FILO). The first item refers to the number of public keys, which as per our requirement, is 4. Hence, the function will proceed to pop out the next 4 items from the stack and store them into a list. Note that the list will contain the public keys in the order it was inserted.



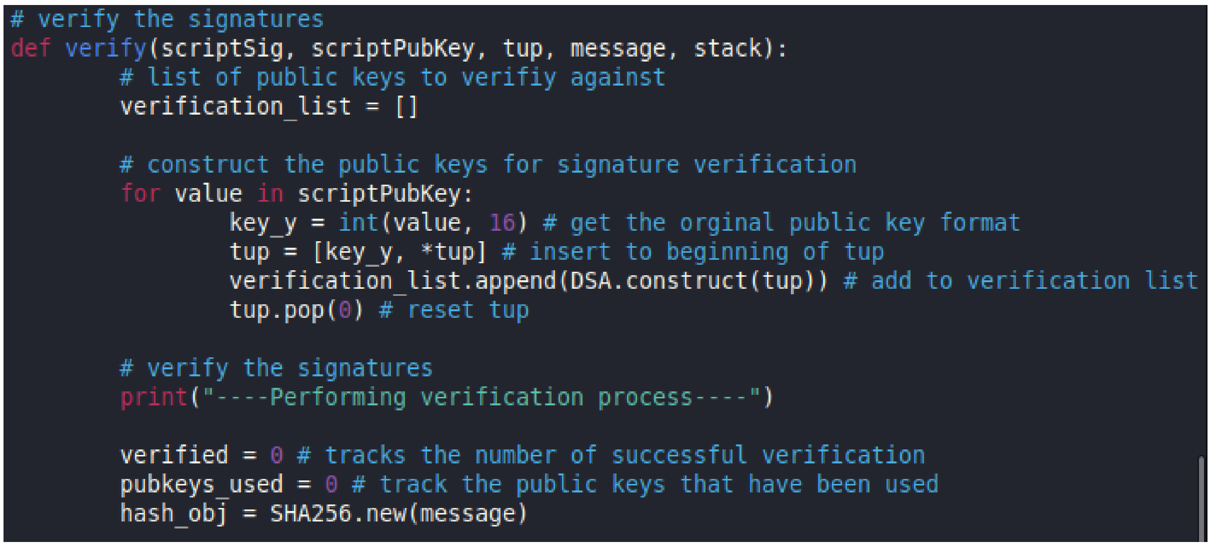
```
# pop the number of signatures
print("--------------------------------------------------")
print("Popping the opcode for the number of signatures...")
print("--------------------------------------------------")
number_of_signatures = int(stack.pop())

# show stack after processing
display_stack(stack)

# create a list to store the signatures
scriptSig = []

# pop the signatures and store them into a list
print("-----------------------")
print(f"Popping {number_of_signatures} signatures...")
print("-----------------------")

# store signatures
for index in range(number_of_signatures):
        scriptSig = [stack.pop(), *scriptSig] # insert as per original order

# show stack after processing
display_stack(stack)

return scriptPubKey, scriptSig, stack
```

- The next item in the stack specifies the number of signatures to capture, which is 2 in this case. The function then proceeds to pop out 2 items from the stack, which are the signatures that will be verified. These signatures will also be stored in a separate list just like the public keys.
- Finally, the function will return the list of public keys and signatures as well as the state of the stack.



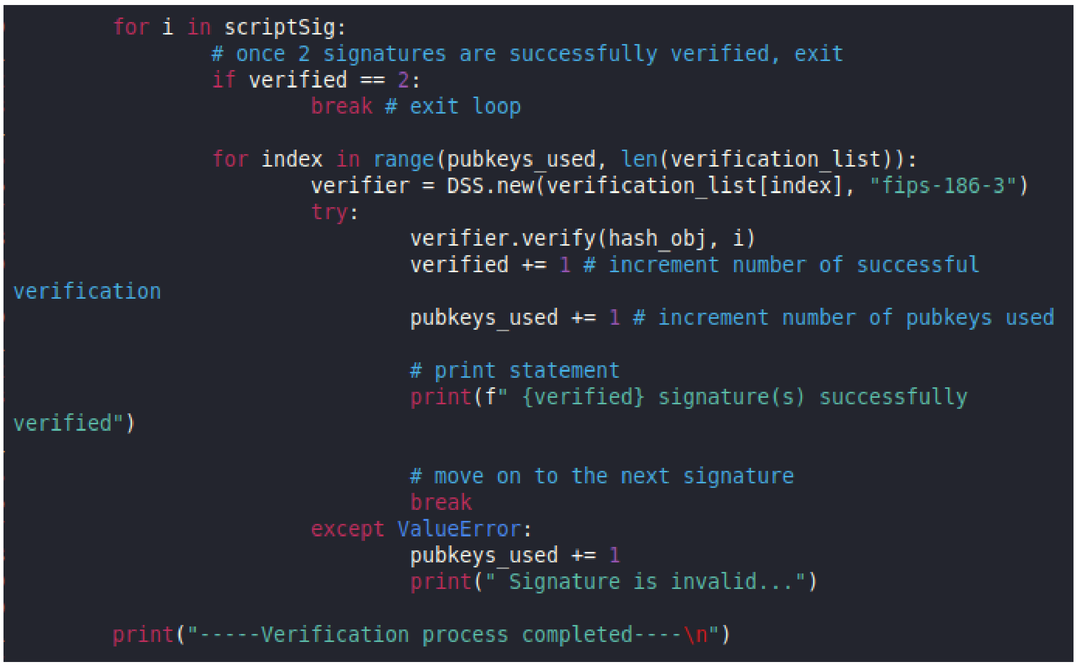
```
        # once all pubkeys and signatures are popped, verify the signatures against the
pubkeys
        stack = verify(scriptSig, scriptPubKey, tup_list[option -1], message, stack)
```

- Once the stack contents have been processed, the signatures will then go through the process of verification.



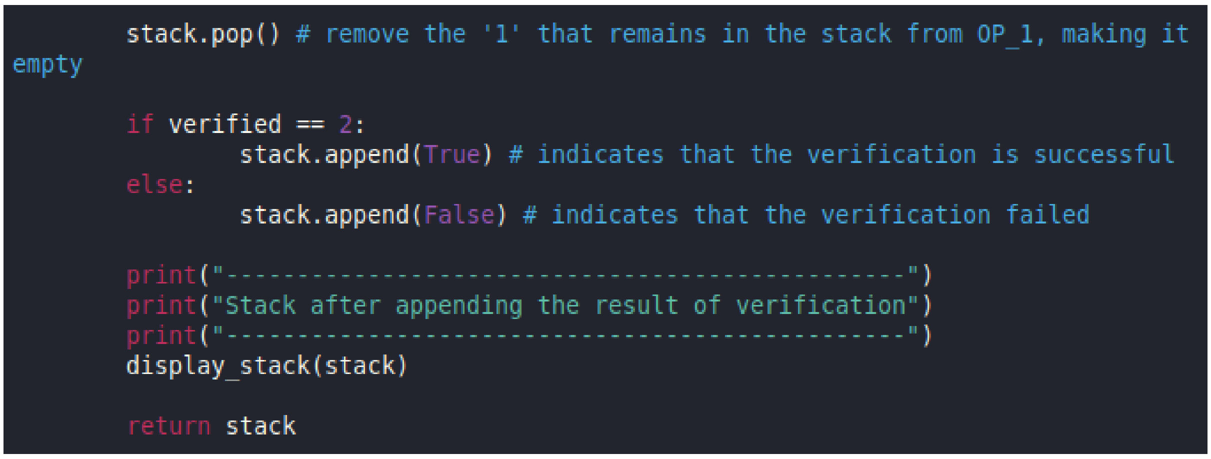
```python
# verify the signatures
def verify(scriptSig, scriptPubKey, tup, message, stack):
    # list of public keys to verifiy against
    verification_list = []

    # construct the public keys for signature verification
    for value in scriptPubKey:
        key_y = int(value, 16) # get the orginal public key format
        tup = [key_y, *tup] # insert to beginning of tup
        verification_list.append(DSA.construct(tup)) # add to verification list
        tup.pop(0) # reset tup

    # verify the signatures
    print("----Performing verification process----")

    verified = 0 # tracks the number of successful verification
    pubkeys_used = 0 # track the public keys that have been used
    hash_obj = SHA256.new(message)
```

- The function above creates an empty list, 'verification_list', which will be used to store all the relevant parameters and information needed for the verification of the signatures.
- To obtain the relevant information mentioned above, the function will go through each of the public key that has been popped out of the stack and use it to construct a public key object, which will then be added to the 'verification_list'.
- The 'verified' is created to track the number of successful signature verification, which will be used to determine whether the verification process is successful or unsuccessful.
- The 'pubkeys_used' variable is created to track the public key objects that have been used in the attempt to verify the signatures. According to the reference site given in the assignment specifications, the public key objects will not be used more than once, regardless of whether it successfully verifies the signature or not. This variable will act as a control to cater to the restrictions above.



```python
        for i in scriptSig:
                # once 2 signatures are successfully verified, exit
                if verified == 2:
                        break # exit loop

                for index in range(pubkeys_used, len(verification_list)):
                        verifier = DSS.new(verification_list[index], "fips-186-3")
                        try:
                                verifier.verify(hash_obj, i)
                                verified += 1 # increment number of successful
verification
                                pubkeys_used += 1 # increment number of pubkeys used

                                # print statement
                                print(f" {verified} signature(s) successfully
verified")

                                # move on to the next signature
                                break
                        except ValueError:
                                pubkeys_used += 1
                                print(" Signature is invalid...")

        print("-----Verification process completed----\n")
```

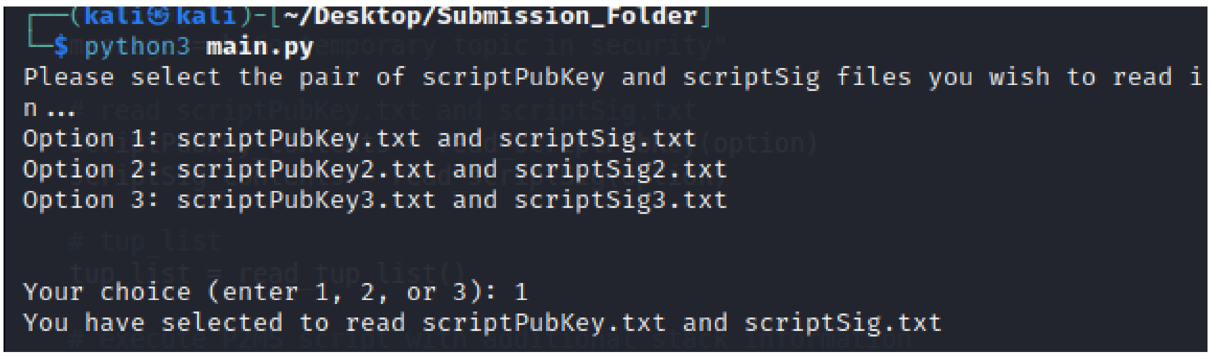
- The verification function then proceeds to verify each signature against the public keys until the required number of signatures have been successfully verified or until the public key objects run out.



```
        stack.pop() # remove the '1' that remains in the stack from OP_1, making it
empty

        if verified == 2:
                stack.append(True) # indicates that the verification is successful
        else:
                stack.append(False) # indicates that the verification failed

        print("------------------------------------------------")
        print("Stack after appending the result of verification")
        print("------------------------------------------------")
        display_stack(stack)

        return stack
```

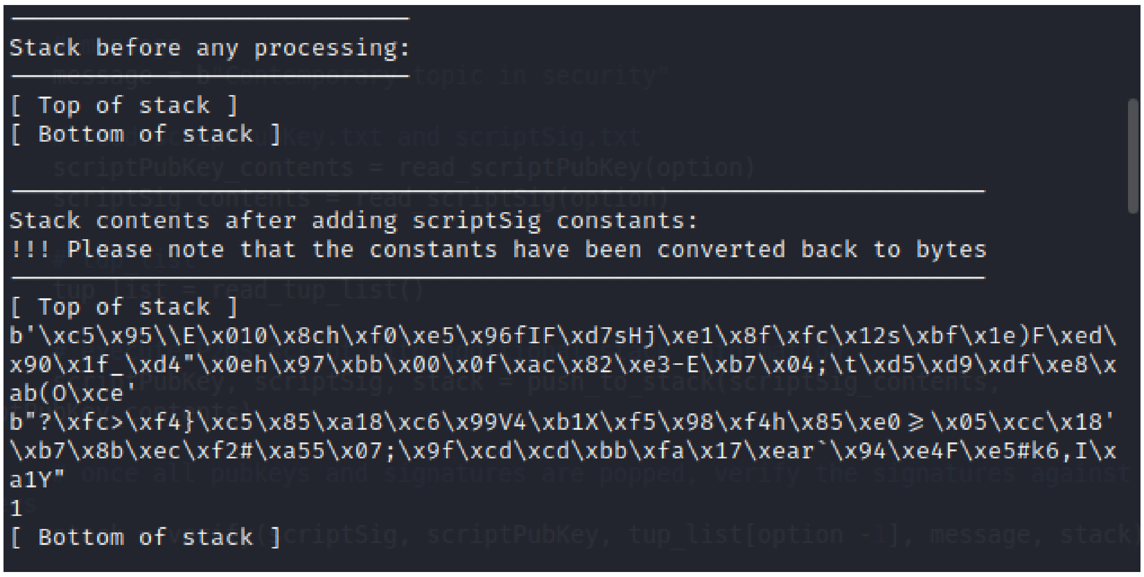
- If the required number of signatures have been successfully verified, the verification process is a success, otherwise, it means that the signatures are invalid.
- Moving on, the function pops the last remaining item in the stack, which is a dummy value, and push the result of the verification process into the stack – 'True' for a successful process, and 'False' for an unsuccessful process.

<u>Running the program – main.py:</u>

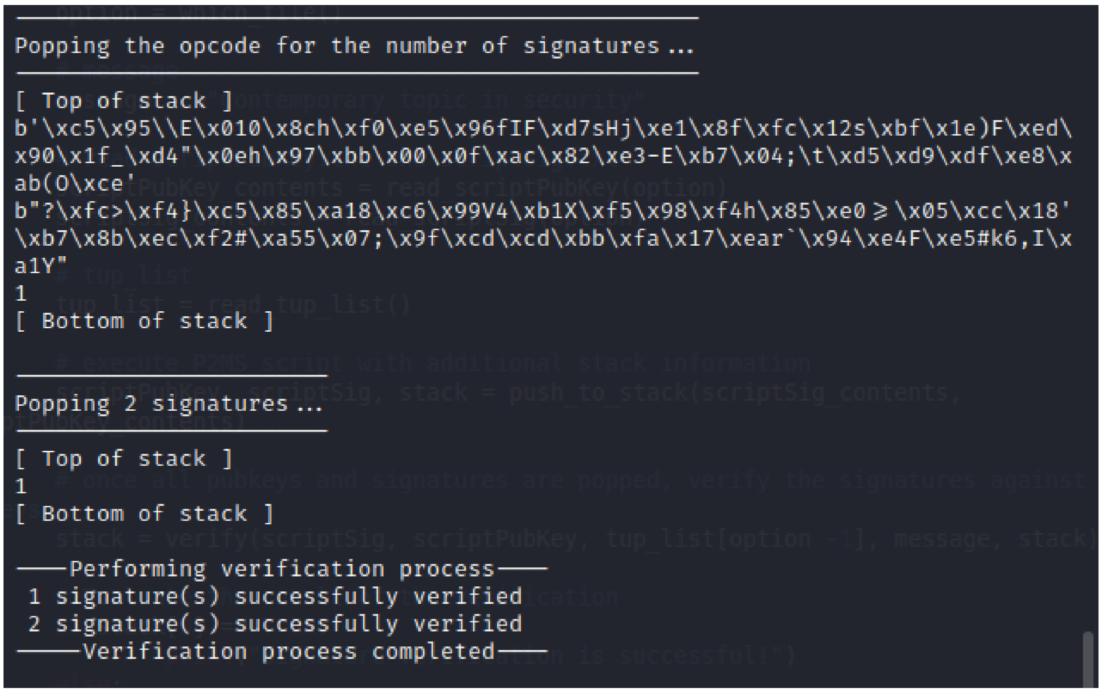


```
┌──(kali㉿kali)-[~/Desktop/Submission_Folder]
└─$ python3 main.py
Please select the pair of scriptPubKey and scriptSig files you wish to read i
n ...
Option 1: scriptPubKey.txt and scriptSig.txt
Option 2: scriptPubKey2.txt and scriptSig2.txt
Option 3: scriptPubKey3.txt and scriptSig3.txt


Your choice (enter 1, 2, or 3): 1
You have selected to read scriptPubKey.txt and scriptSig.txt
```

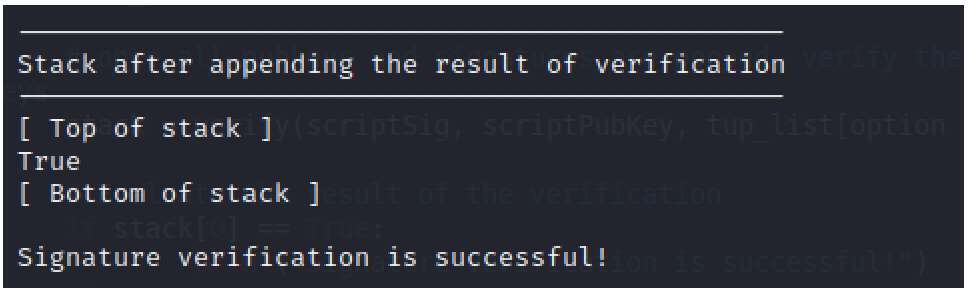
- The program first asks the user for the pair of files the user wishes to use, in the above scenario, I input '1' because I want to read 'scriptPubKey.txt' and 'scriptSig.txt'.
- Below are some of the screenshots of the output from running the program, which shows the state of the stack to aid in the visualisation of the process.



```
───────────────────────────
Stack before any processing:
───────────────────────────
[ Top of stack ]
[ Bottom of stack ]

──────────────────────────────────────────────────────────────────
Stack contents after adding scriptSig constants:
!!! Please note that the constants have been converted back to bytes
──────────────────────────────────────────────────────────────────
[ Top of stack ]
b'\xc5\x95\\E\x010\x8ch\xf0\xe5\x96fIF\xd7sHj\xe1\x8f\xfc\x12s\xbf\x1e)F\xed\
x90\x1f_\xd4"\x0eh\x97\xbb\x00\x0f\xac\x82\xe3-E\xb7\x04;\t\xd5\xd9\xdf\xe8\x
ab(O\xce'
b"?\xfc>\xf4}\xc5\x85\xa18\xc6\x99V4\xb1X\xf5\x98\xf4h\x85\xe0≥\x05\xcc\x18'
\xb7\x8b\xec\xf2#\xa55\x07;\x9f\xcd\xcd\xbb\xfa\x17\xear`\x94\xe4F\xe5#k6,I\x
a1Y"
1
[ Bottom of stack ]
```



```
──────────────────────────────────────────────
Popping the opcode for the number of signatures ...
──────────────────────────────────────────────
[ Top of stack ]
b'\xc5\x95\\E\x010\x8ch\xf0\xe5\x96fIF\xd7sHj\xe1\x8f\xfc\x12s\xbf\x1e)F\xed\
x90\x1f_\xd4"\x0eh\x97\xbb\x00\x0f\xac\x82\xe3-E\xb7\x04;\t\xd5\xd9\xdf\xe8\x
ab(O\xce'
b"?\xfc>\xf4}\xc5\x85\xa18\xc6\x99V4\xb1X\xf5\x98\xf4h\x85\xe0>=\x05\xcc\x18'
\xb7\x8b\xec\xf2#\xa55\x07;\x9f\xcd\xcd\xbb\xfa\x17\xear`\x94\xe4F\xe5#k6,I\x
a1Y"
1
[ Bottom of stack ]

─────────────────────
Popping 2 signatures ...
─────────────────────
[ Top of stack ]
1
[ Bottom of stack ]

───Performing verification process───
 1 signature(s) successfully verified
 2 signature(s) successfully verified
────Verification process completed───
```



```
──────────────────────────────────────────
Stack after appending the result of verification
──────────────────────────────────────────
[ Top of stack ]
True
[ Bottom of stack ]

Signature verification is successful!
```