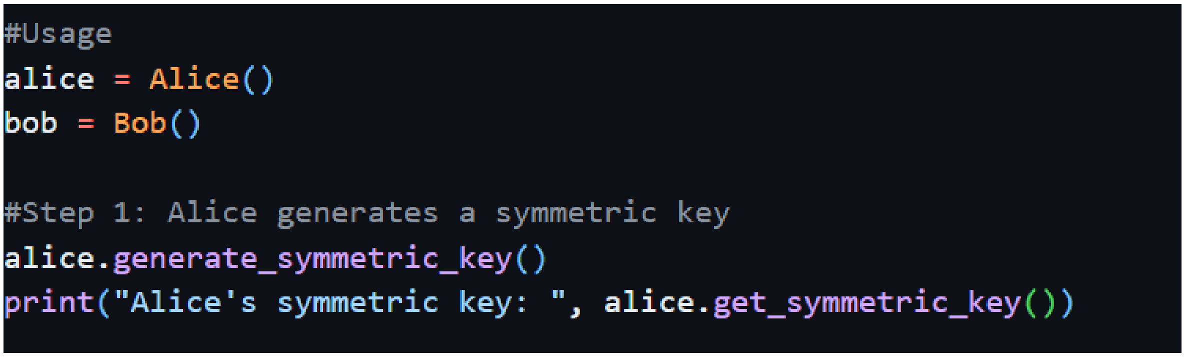
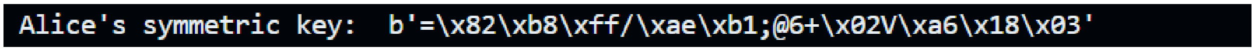
#Step 1: Alice generates a symmetric key... 1
#Step 2: Bob generates a public and private key using RSA..2
#Step 3: Alice encrypts the symmetric key using Bob's public key.......................................3
#Step 4: Bob decrypts the symmetric key using his private key...4
#Step 5: Bob encrypts a message using the symmetric key.. 5
#Step 6: Alice decrypts the message using the symmetric key.. 5

#Step 1: Alice generates a symmetric key



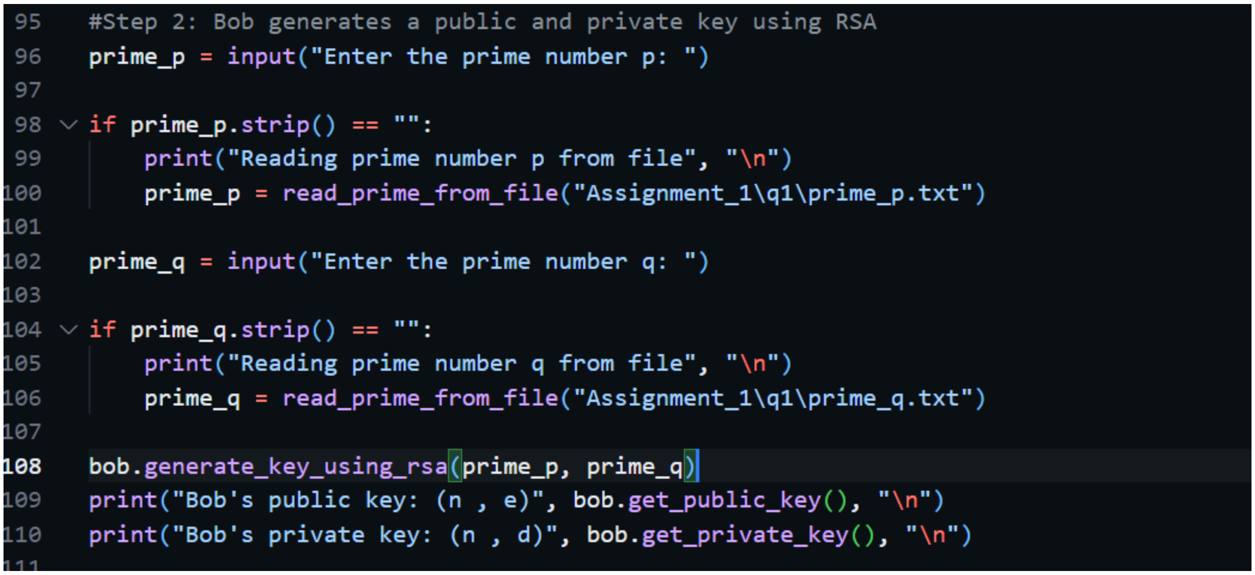
```
#Usage
alice = Alice()
bob = Bob()

#Step 1: Alice generates a symmetric key
alice.generate_symmetric_key()
print("Alice's symmetric key: ", alice.get_symmetric_key())
```

Alice generates a random symmetric key using urandom(16)



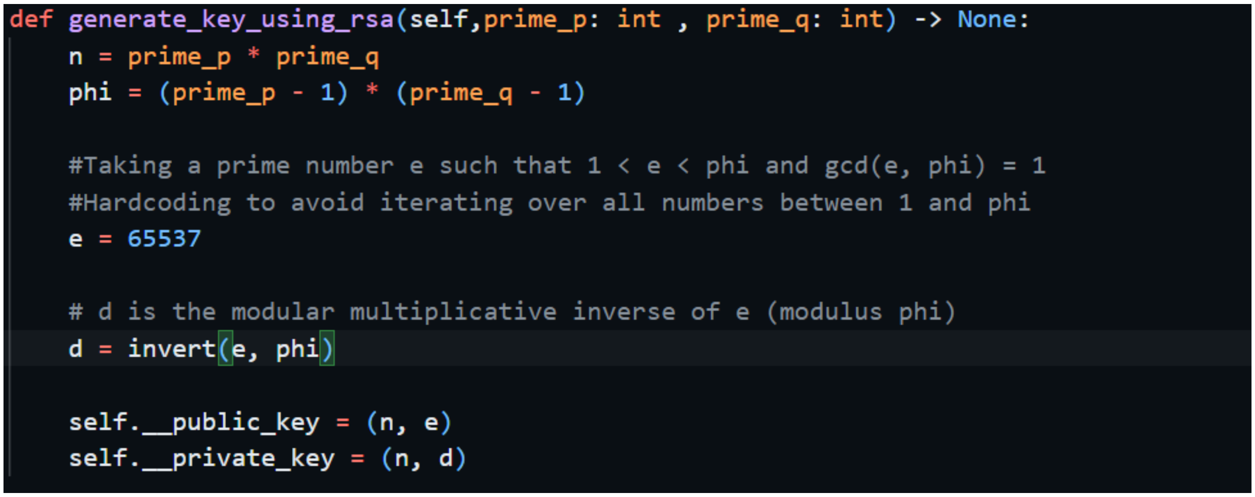
```
Alice's symmetric key:  b'=\x82\xb8\xff/\xae\xb1;@6+\x02V\xa6\x18\x03'
```

#Step 2: Bob generates a public and private key using RSA



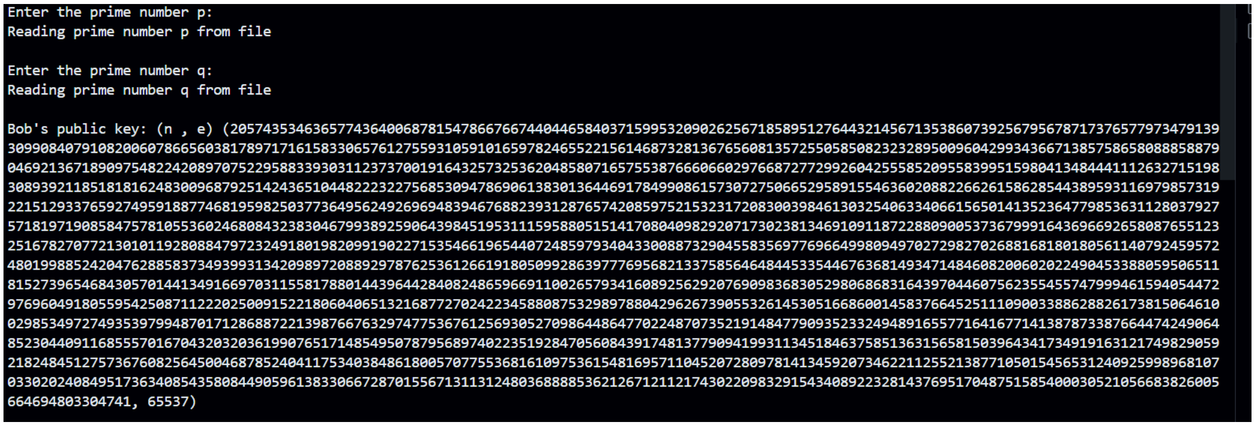
```
#Step 2: Bob generates a public and private key using RSA
prime_p = input("Enter the prime number p: ")

if prime_p.strip() == "":
    print("Reading prime number p from file", "\n")
    prime_p = read_prime_from_file("Assignment_1\q1\prime_p.txt")

prime_q = input("Enter the prime number q: ")

if prime_q.strip() == "":
    print("Reading prime number q from file", "\n")
    prime_q = read_prime_from_file("Assignment_1\q1\prime_q.txt")

bob.generate_key_using_rsa(prime_p, prime_q)
print("Bob's public key: (n , e)", bob.get_public_key(), "\n")
print("Bob's private key: (n , d)", bob.get_private_key(), "\n")
```

As the prime numbers are large, i read them from a file if no user input is given.

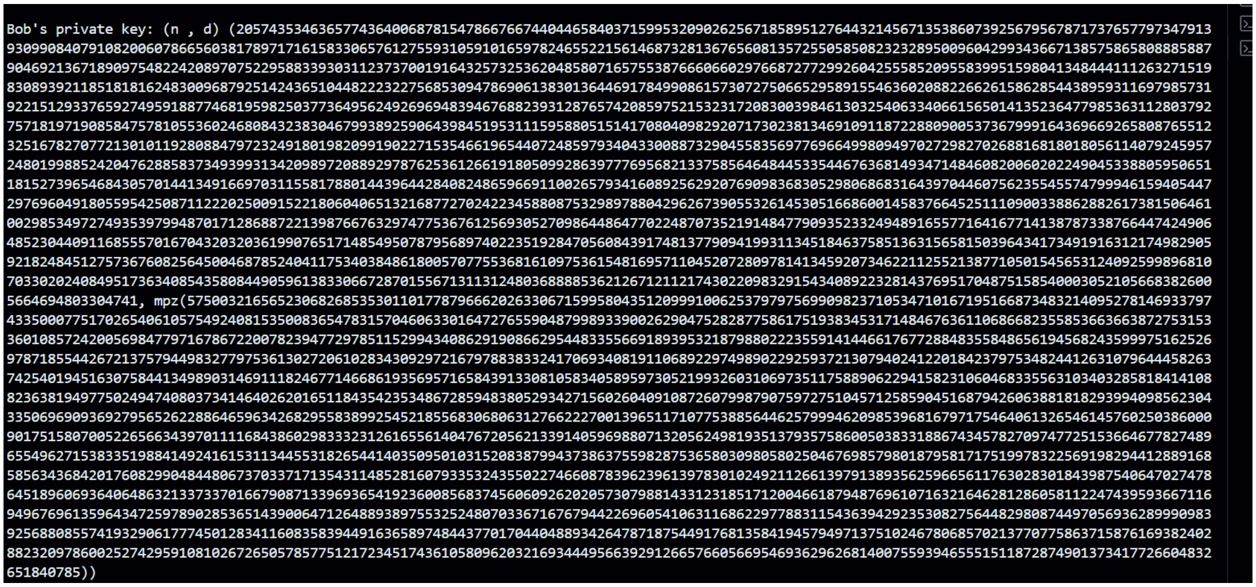


```
def generate_key_using_rsa(self,prime_p: int , prime_q: int) -> None:
    n = prime_p * prime_q
    phi = (prime_p - 1) * (prime_q - 1)

    #Taking a prime number e such that 1 < e < phi and gcd(e, phi) = 1
    #Hardcoding to avoid iterating over all numbers between 1 and phi
    e = 65537

    # d is the modular multiplicative inverse of e (modulus phi)
    d = invert(e, phi)

    self.__public_key = (n, e)
    self.__private_key = (n, d)
```

RSA algorithm uses hard coded 'e' to save time.

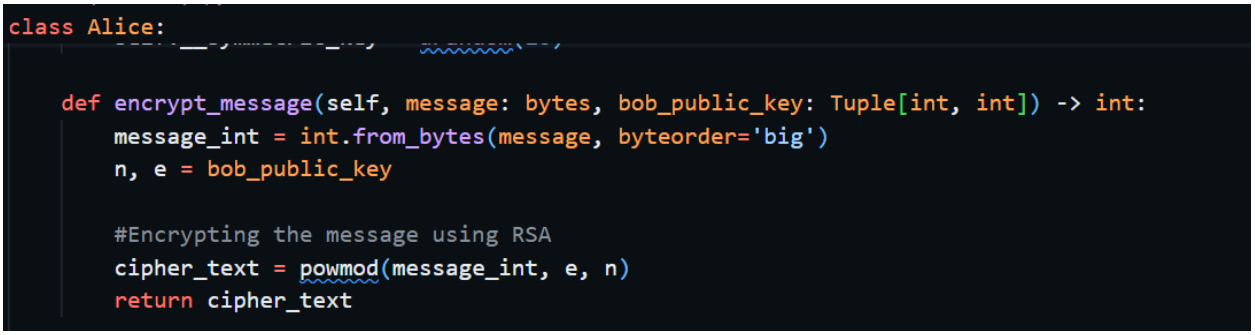


```
Enter the prime number p:
Reading prime number p from file

Enter the prime number q:
Reading prime number q from file

Bob's public key: (n , e) (20574353463657743640068781547866766744044658403715995320902625671858951276443214567135386073925679567871737657779734791 39
309908407910820060786656038178971716158330657612755931059101659782465522156146873281367656081357255058508232328950096042993436671385758658088858879
046921367189097548224208970752295883393031123737001916432573253620485807165755387666066029766872772992604255585209558399515980413484441112632715198
308939211851818162483009687925142436510448222322756853094786906138301364469178499086157307275066529589155463602088226626158628544389593116979857319
221512933765927495918877468195982503773649562492696948394676882393128765742085975215323172083003984613032540633406615650141352364779853631128037927
571819719085847578105536024680843238304679938925906439845195311159588051514170804098292071730238134691091187228809005373679991643696692658087655123
251678270772130101192808847972324918019820991902271535466196544072485979340433008873290455835697769664998094970272982702688168180180561140792459572
480199885242047628858373493993134209897208892978762536126619180509928639777695682133758564648445335446763681493471484608200602022490453388059506511
815273965468430570144134916697031155817880144396442840824865966911002657934160892562920769098368305298068683164397044607562355455747999461594054472
976960491805595425087112220250091522180604065132168772702422345880875329897880429626739055326145305166860014583766452511109003388628826173815064610
029853497274935397994870171286887221398766763297477536761256930527098644864770224870735219148477909352332494891655771641677141387873387664474249064
852304409116855570167043203203619907651714854950787956897402235192847056084391748137790941993113451846375851363156581503964341734919163121749829059
218248451275736760825645004687852404117534038486180057077553681610975361548169571104520728097814134592073462211255213877105015456531240925998968107
033020240849517363408543580844905961383306672870155671311312480368888536212671211217430220983291543408922328143769517048751585400030521056683826005
664694803304741, 65537)
```

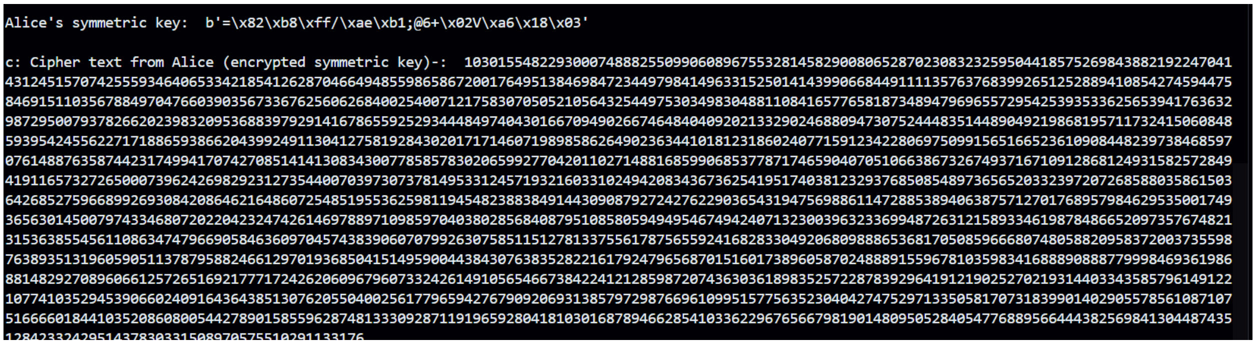


```
Bob's private key: (n , d) (2057435346365774364006878154786676674404465840371599532090262567185895127644321456713538607392567956787173765779734791 3
930990840791082006078665603817897171615833065761275593105910165978246552215614687328136765608135725505850823232895009604299343667138575865808885887
904692136718909754822420897075229588339303112373700191643257325362048580716575538766606602976687277299260425558520955839951598041348444111263271519
830893921185181816248300968792514243651044822232275685309478690613830136446917849908615730727506652958915546360208822662615862854438959311697985731
922151293376592749591887746819598250377364956249269694839467688239312876574208597521532317208300398461303254063340661565014135236477985363112803792
757181971908584757810553602468084323830467993892590643984519531115958805151417080409829207173023813469109118722880900537367999164369669265808765512
325167827077213010119280884797232491801982099190227153546619654407248597934043300887329045583569776966499809497027298270268816818018056114079245957
248019988524204762885837349399313420989720889297876253612661918050992863977769568213375856464844533544676368149347148460820060202249045338805950651
181527396546843057014413491669703115581788014439644284082486596691100265793416089256292076909836830529806868316439704460756235545574799946159405447
297696049180559542508711222025009152218060406513216877270242234588087532989788042962673905532614530516686001458376645251110900338862882617381506461
002985349727493539799487017128688722139876676329747753676125693052709864486477022487073521914847790935233249489165577164167714138787338766447424906
485230440911685557016704320320361990765171485495078795689740223519284705608439174813779094199311345184637585136315658150396434173491916312174982905
921824845127573676082564500468785240411753403848618005707755368161097536154816957110452072809781413459207346221125521387710501545653124092599896810
703302024084951736340854358084490596138330667287015567131131248036888853621267121121743022098329154340892232814376951704875158540003052105668382600
5664694803304741, mpz(575003216565230682685353011017787966620263306715995804351209991006253797975699098237105347101671951668734832140952781469337 97
433500077517026540610575492408153500836547831570460633016472765590487998933900262904752828775861751938345317148467636110686682355853663663872753153
360108572420056984779716786722007823947729785115299434086291908662954483355669189395321879880222355914144661767728848355848656194568243599975162526
978718554426721375794498327797536130272061028343092972167978838332417069340819110689229749890229259372130794024122018423797534824412631079644458263
742540194516307584413498903146911182467714668619356957165843913308105834058959730521993260310697351175889062294158231060468335563103403285818414108
823638194977502494740803734146402620165118435423534867285948380529342715602604091087260799879075972751045712585904516879426063881818293994098562304
335069690936927956526228864659634268295583899254521855683068063127662227001396511710775388564462579994620985396816797175464061326546145760250386000
901751580700522656634397011116843860298333231261655614047672056213391405969880713205624981935137935758600503833188674345782709747725153664677827489
655496271538335198841492416153113445531826544140350950103152083879943738637559828753658030980580250467698579801879581717519978322569198294412889168
585634368420176082990484480673703371713543114852816079335324355022746608783962396139783010249211266139791389356259665611763028301843987540647027478
645189606936406486321337337016679087133969365419236008568374560609262020573079881433123185171200466187948769610716321646281286058112247439593667116
949676961359643472597890285365143900647126488938975532524807033671676794422696054106311686229778831154363942923530827564482980874497056936289990983
925688085574193290617774501283411608358394491636589748443770170440488934264787187544917681358419457949713751024678068570213770775863715876169382402
882320978600252742959108102672650578577512172345174361058096203216934449566392912665766056695469362962681400755939465551511872874901373417726604832
651840785))
```

#Step 3: Alice encrypts the symmetric key using Bob's public key

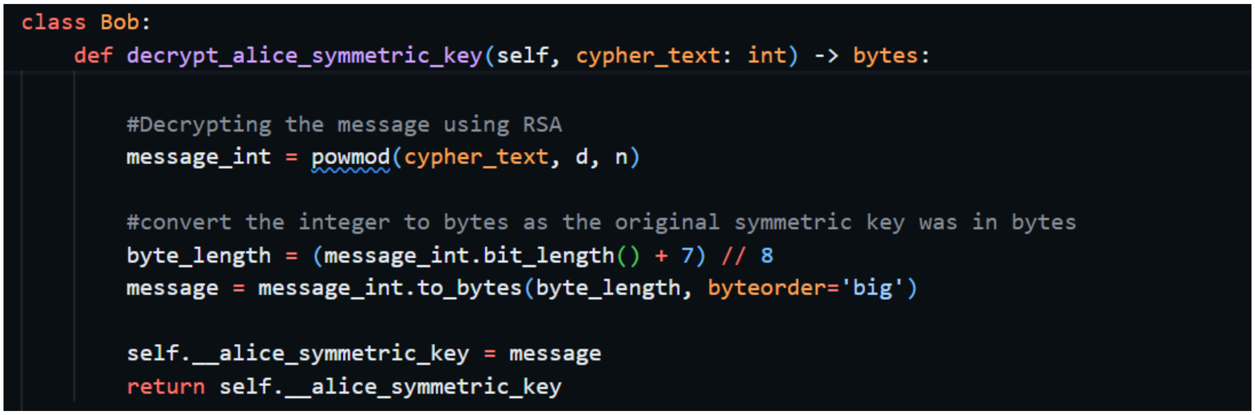


```
class Alice:

    def encrypt_message(self, message: bytes, bob_public_key: Tuple[int, int]) -> int:
        message_int = int.from_bytes(message, byteorder='big')
        n, e = bob_public_key

        #Encrypting the message using RSA
        cipher_text = powmod(message_int, e, n)
        return cipher_text
```

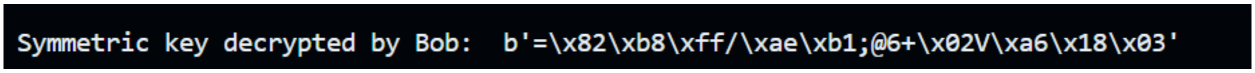


```
Alice's symmetric key:  b'=\x82\xb8\xff/\xae\xb1;@6+\x02V\xa6\x18\x03'

c: Cipher text from Alice (encrypted symmetric key)-:  10301554822930007488825509906089675532814582900806528702308323259504418575269843882192247041
431245157074255593464065334218541262870466494855986586720017649513846984723449798414963315250141439906684491111357637683992651252889410854274594475
846915110356788497047660390356733676256062684002540071217583070505210564325449753034983048811084165776581873489479696557295425393533625653941763632
987295007937826620239832095368839792914167865592529344484974043016670949026674648404092021332902468809473075244483514489049219868195711732415060848
593954245562271718865938662043992491130412758192843020171714607198985862649023634410181231860240771591234228069750991565166523610908448239738468597
076148876358744231749941707427085141413083430077858578302065992770420110271488168599068537787174659040705106638673267493716710912868124931582572849
419116573272650007396242698292312735440070397307378149533124571932160331024942083436736254195174038123293768508548973656520332397207268588035861503
642685275966899269308420864621648607254851955362598119454823883849144309087927242762290365431947569886114728853894063875712701768957984629535001749
365630145007974334680720220423247426146978897109859704038028568408795108580594949546749424071323003963233699487263121589334619878486652097357674821
315363855456110863474796690584636097045743839060707992630758511512781337556178756559241682833049206809888653681705085966680748058820958372003735598
763893513196059051137879588246612970193685041514959004438430763835282216179247965687015160173896058702488891559678103598341688890888779998469361986
881482927089606612572651692177717242620609679607332426149105654667384224121285987207436303618983525722878392964191219025270219314403343585796149122
107741035294539066024091643643851307620550400256177965942767909206931385797298766961099515775635230404274752971335058170731839901402905578561087107
516666018441035208608005442789015855962874813330928711919659280418103016878946628541033622967656679819014809505284054776889566444382569841304487435
1284233242951437830331508970575510291133176
```

#Step 4: Bob decrypts the symmetric key using his private key

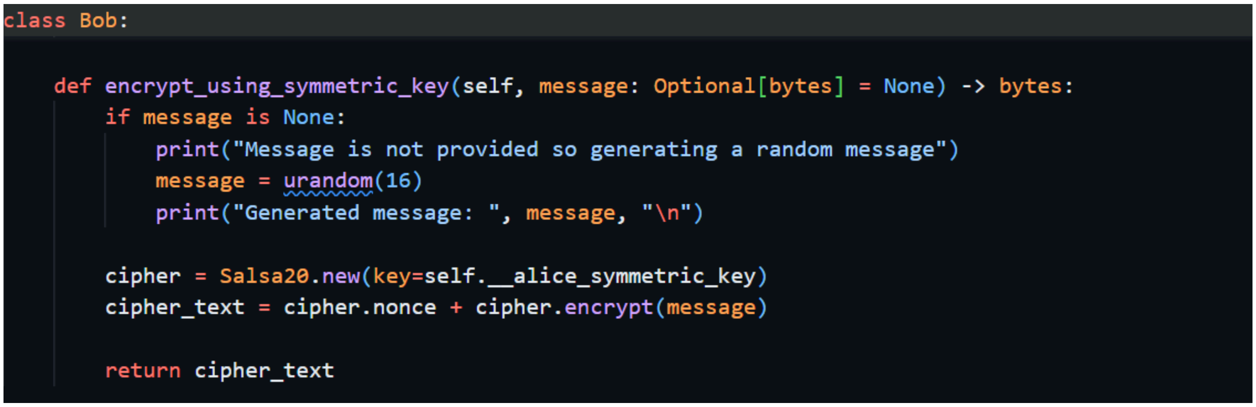


```
class Bob:
    def decrypt_alice_symmetric_key(self, cypher_text: int) -> bytes:

        #Decrypting the message using RSA
        message_int = powmod(cypher_text, d, n)

        #convert the integer to bytes as the original symmetric key was in bytes
        byte_length = (message_int.bit_length() + 7) // 8
        message = message_int.to_bytes(byte_length, byteorder='big')

        self.__alice_symmetric_key = message
        return self.__alice_symmetric_key
```

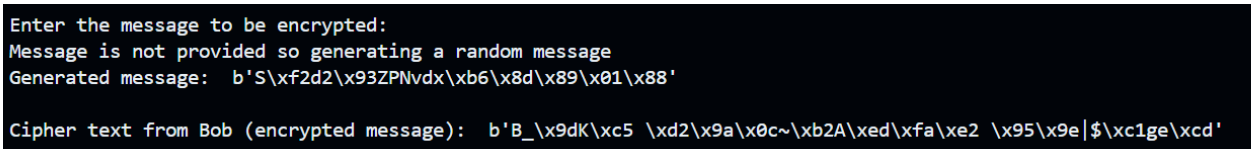


```
Symmetric key decrypted by Bob:  b'=\x82\xb8\xff/\xae\xb1;@6+\x02V\xa6\x18\x03'
```

#Step 5: Bob encrypts a message using the symmetric key

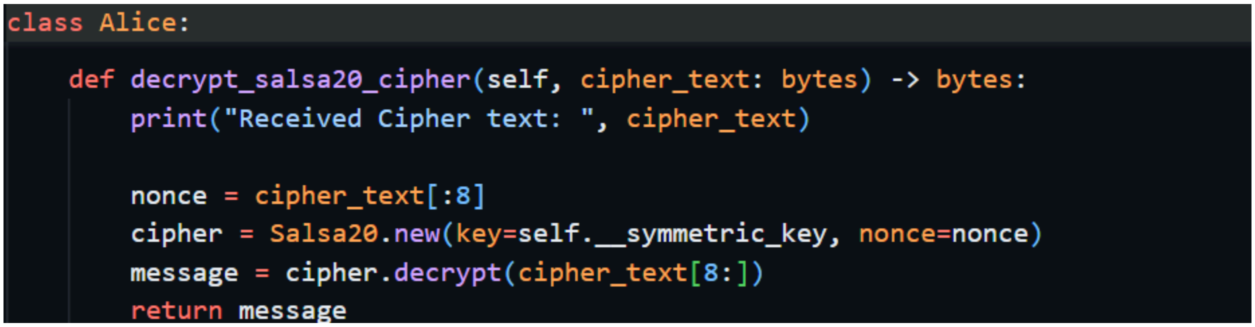


```
class Bob:

    def encrypt_using_symmetric_key(self, message: Optional[bytes] = None) -> bytes:
        if message is None:
            print("Message is not provided so generating a random message")
            message = urandom(16)
            print("Generated message: ", message, "\n")

        cipher = Salsa20.new(key=self.__alice_symmetric_key)
        cipher_text = cipher.nonce + cipher.encrypt(message)

        return cipher_text
```



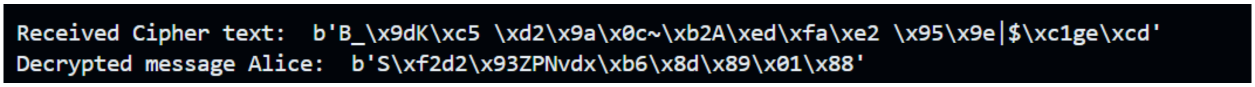
```
Enter the message to be encrypted:
Message is not provided so generating a random message
Generated message:  b'S\xf2d2\x93ZPNvdx\xb6\x8d\x89\x01\x88'

Cipher text from Bob (encrypted message):  b'B_\x9dK\xc5 \xd2\x9a\x0c~\xb2A\xed\xfa\xe2 \x95\x9e|$\xc1ge\xcd'
```

#Step 6: Alice decrypts the message using the symmetric key



```
class Alice:

    def decrypt_salsa20_cipher(self, cipher_text: bytes) -> bytes:
        print("Received Cipher text: ", cipher_text)

        nonce = cipher_text[:8]
        cipher = Salsa20.new(key=self.__symmetric_key, nonce=nonce)
        message = cipher.decrypt(cipher_text[8:])
        return message
```

Decrypts using Salsa20



```
Received Cipher text:  b'B_\x9dK\xc5 \xd2\x9a\x0c~\xb2A\xed\xfa\xe2 \x95\x9e|$\xc1ge\xcd'
Decrypted message Alice:  b'S\xf2d2\x93ZPNvdx\xb6\x8d\x89\x01\x88'
```