

**Secure Communications Module**

**Python Coding Labs based Report**

**James Finglas – B00094138**

Department of Informatics

School of Informatics and Engineering

Institute of Technology Blanchardstown

Dublin 15.

Word count:

**BN311 Bachelor of Science in Computing**

**Digital Forensics & Cyber Security**

**?/5/2019**

Contents

[1: Brute Force Ciphers 3](#_Toc4401149)

[1.1: Rot5 3](#_Toc4401150)

[1.2: Rot13 5](#_Toc4401151)

[1.3: Rot47 7](#_Toc4401152)

[2: Credit Card Validate 8](#_Toc4401153)

[3: Simple Blockchain 13](#_Toc4401154)

[4: RSA Challenges 15](#_Toc4401155)

[4.1: RSA Challenge 1 15](#_Toc4401156)

[4.2: RSA Challenge 2 16](#_Toc4401157)

[4.3 RSA Challenge 3 17](#_Toc4401158)

[4.4: RSA Challenge 4 19](#_Toc4401159)

[4.5 RSA challenge 5 20](#_Toc4401160)

[4.6 RSA challenge 6 21](#_Toc4401161)

[4.7 RSA Challenge 7 23](#_Toc4401162)

[4.8 RSA Challenge 8 24](#_Toc4401163)

[4.9 RSA Challenge 9 25](#_Toc4401164)

[4.10 RSA Challenge 10 26](#_Toc4401165)

[4.11 RSA Challenge 11 26](#_Toc4401166)

[References: 26](#_Toc4401167)

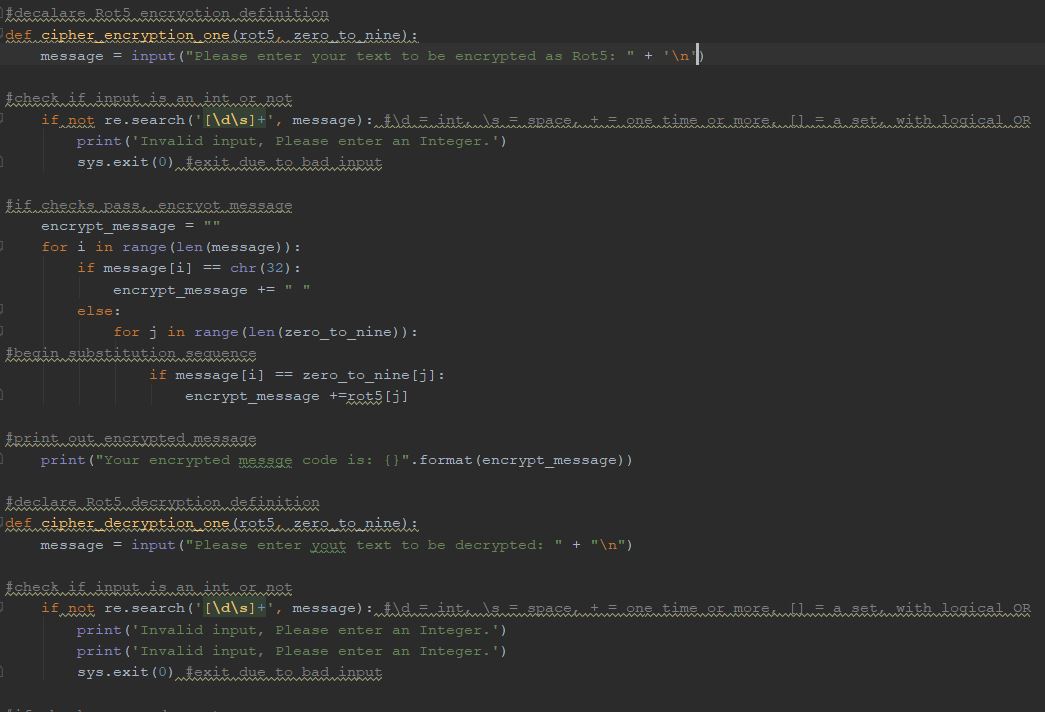
[Appendix 26](#_Toc4401168)

# 1: Brute Force Ciphers

Our task for this first coding lab is create a program which can brute force all combinations of a given user input via 3 methods, Rot5, Rot13 and rot47.

### 1.1: Rot5

I chose to implement a simple menu system that would allow a user to select their preferred decryption type. Below you can see my Rot5 decryption algorithm.

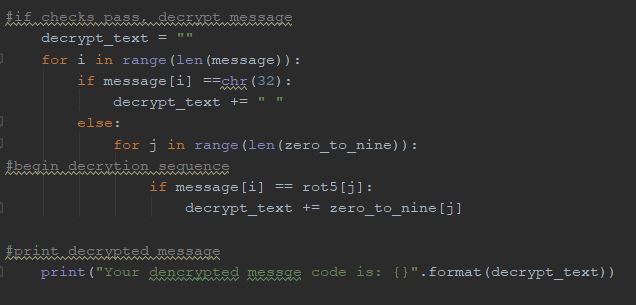


Once input is taken in, it is verified see if the input is valid in order to decrypt anything via Rot5 the input must contain integers. Rot5 cannot decrypt Characters.

Should the input pass the first validation check, it is passed through a loop and evaluated against the first of the alphabets created in our main method as seen here:



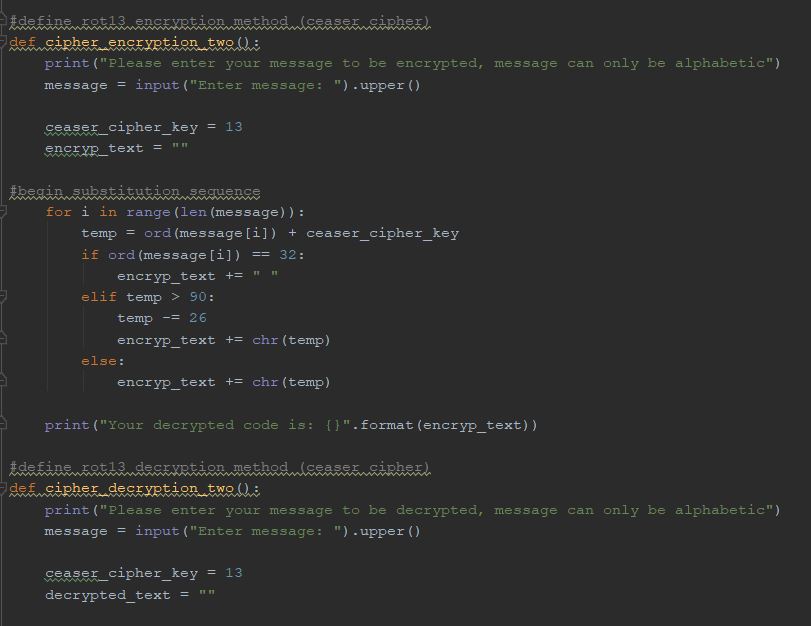
Finally, the text is passed through another loop to be evaluated against the second of the alphabets we declared in our main method.



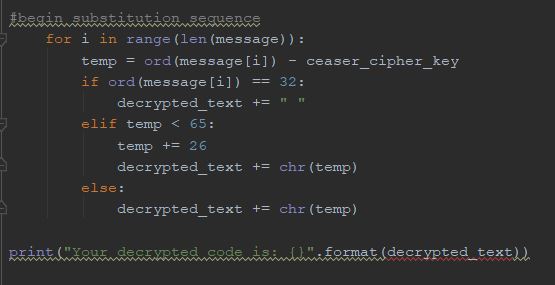
The output then gives us all possible iterations via Rot5, starting and ending with our given input.

### 1.2: Rot13

Rot13, or the Caesar Cipher, proved slightly easier to implement as we learned how to implement and use alphabets from implementing Rot5. As we can see below, we have no declared our alphabet within our function. The entire process is a simple ascii check. Firstly, we pass the characters of the input into a loop and evaluate them to see if the int of the characters is equal to 32. If it is, the character is a space and is appended to the message as a space.



If the input is not a space, it is evaluated to see if is first greater then 90. Is so, we subtract 26 from it and append the ascii equivalent of the output in int form to the string.

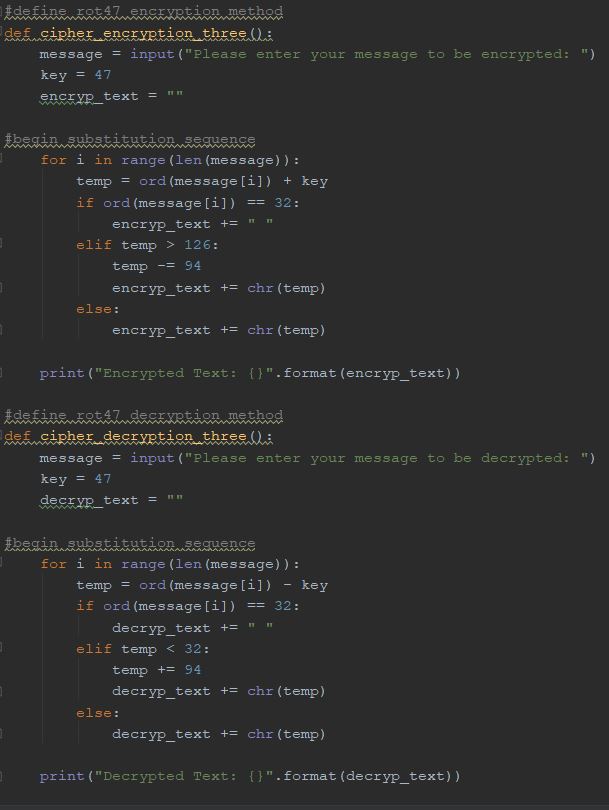


If the character is less than 65, we add 26 to it, and append the ascii equivalent of the new int to the string. This gives us all of the iterations possible in Rot13.

### 1.3: Rot47

Finally, we have Rot47, which proved to be the easiest to implement as the alphabet contains all possible combinations of letters both upper and lower case, special characters and numbers.

This means the evaluation check and maths are simple. First, we check as before to see it the input is 32 and append a space to the string if it is. If not, we simply evaluate if the input is greater then 126. If so we subtract 94 from it and append the ascii equivalent of the sum to the string. Lastly, we check if the number is less than 32, and if it is, we add 94 to the number and append it to the string. This gives us all possible iterations as loop through the array of characters in Rot47.



# 2: Credit Card Validate

We are given 4 tasks for this lab that combine to create a fully functional credit card validator. Two of these tasks are relatively simple and two are more complex. The tasks are as follows:

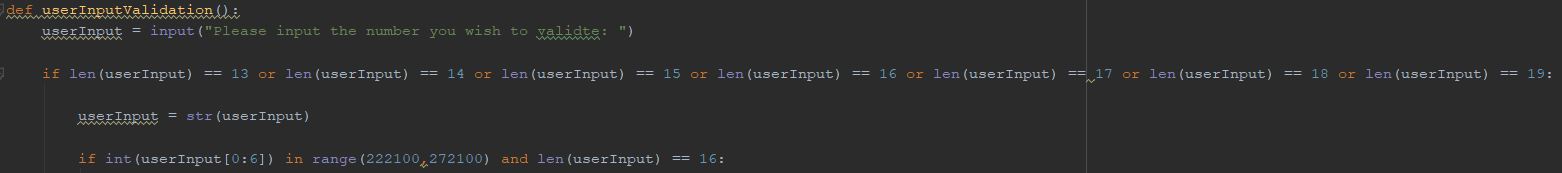
1: Accept user input and validate if this input is a valid credit number. (I elected to use only valid numbers based upon the website provided by our lecture (<https://www.freeformatter.com/credit-card-number-generator-validator.html>).

2: Accept user input and validate if this credit number matches one of a given series of vendors. Again, I used the vendors listed on the page listed above.

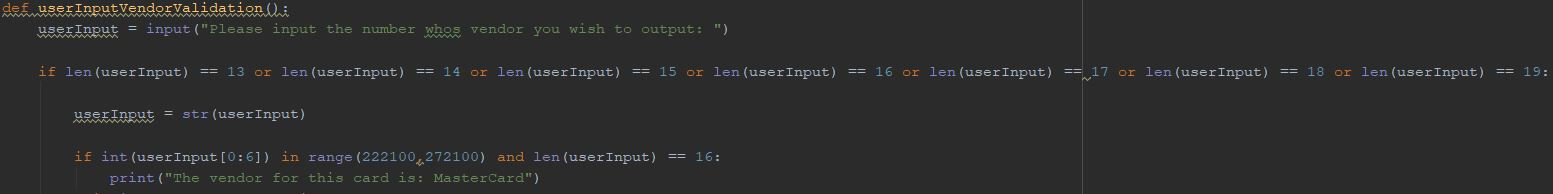
3: Accept user input and perform a checksum calculation on it. In this case the input shall only be 12 characters long and we must generate the appropriate number of characters such that it validates. I elected to also validate all checksums between the ranges of 14-19 in line with the correct of the card lengths listed on the website listed above.

4: Accept user input and based on that input, generate a valid credit number.

Two of these functions are achieve using basic if statements in my program. These programs accept user input based on a simple menu system, should the user input a credit number that is within the right range of starting digits, and be within the correct length as per the option selected, the card is validated as being a correct credit card number, otherwise it is invalid.



Here we can see an example as explained above. In this case should the user enter a string that is within the range of 14-19 characters in length it is evaluated, and should the first 5 digits be within the range of 222100-272-099, the card is considered a valid number. There are far too many if statements to list in the report but they all follow this format.

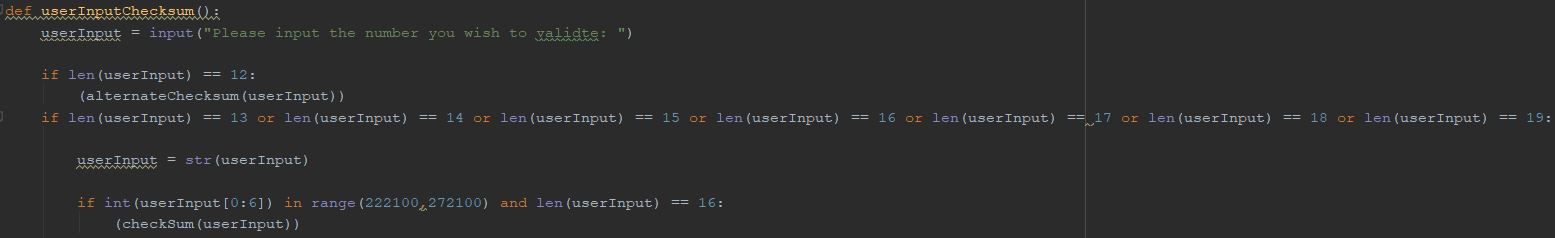


Here we can see the exact same format implemented for the vendor checking function. If the input is within the given length, and the first 5 characters are within the given range, the card is considered a valid MasterCard.

While the input validation is the same for the checksum and generate functions, they require other functions to be created that are a little more complicated. The checksum function is based on the ‘Luhn’ mathematical formula. The concept of the formula is simple. First, we must drop the last digit from the number. The last digit is what we want to check against. Next, we reverse the numbers. Then, we multiply the digits in odd positions (1, 3, 5, etc.) by 2 and subtract 9 to all any result higher than 9. Next, we add all the numbers together.

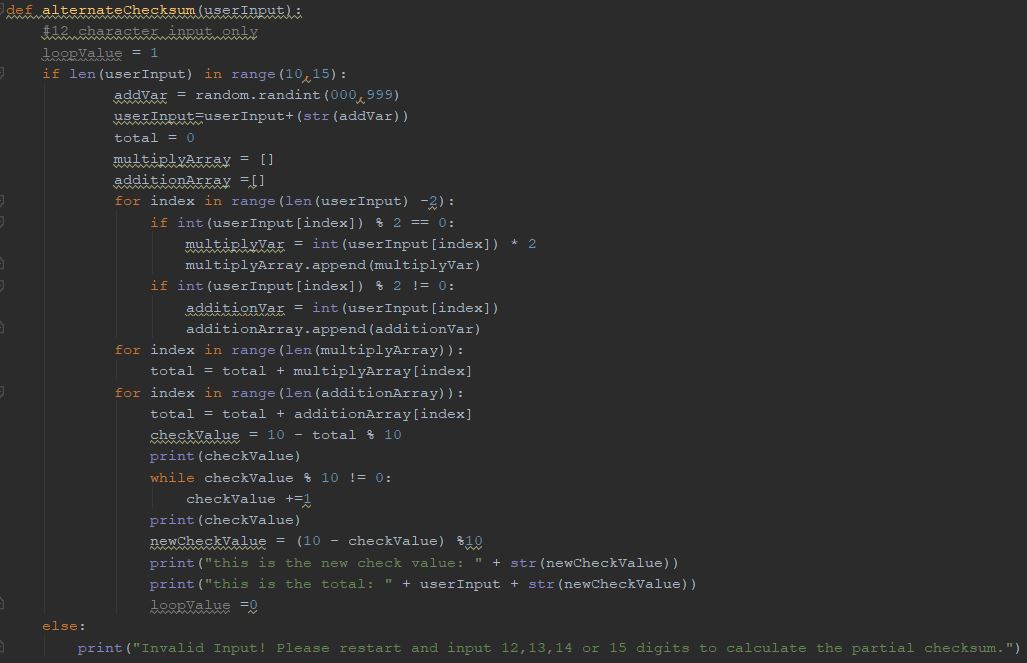
Finally, we get the modulus 10 value of this value. We compare this value to the check digit, which is always the last number of the credit card. If this modulus 10 value is equal to the check number (last digit of credit card) we have confirmed a valid credit card number.

However, in practice when coding it is actually even simpler. If you add up all the numbers of the card having removed the check digit for comparison later, then reverse the digits and add them up again, the result is the same. So, in practice we do not need to reverse the sequence at all. Below we can see my checksum function.



First, I created a variation of previously shown simple if validation statements. This one validates the input is correct, and if the string is 12 characters in length it will call my alternative checksum function which shall generate 3 random digits, finally it will attempt to checksum verify the new number and it will repeat this cycle, until it generates the appropriate number that when added to the current number, will result in a valid checksum. Finally, this number is output.

Here we have the alternative checksum function:



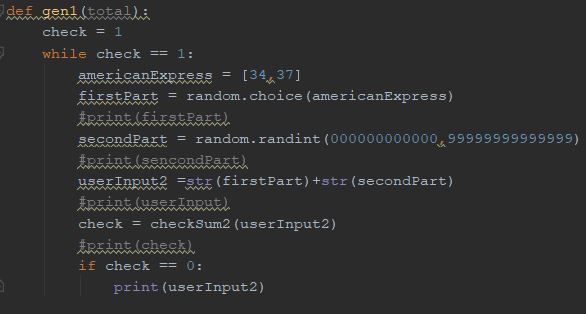
This function takes in the user input, generates 3 random integers and appends them to the input. Next, it passed this new 15-digit string through a loop. It checks each digit in the array to see if the modulus by 2 of this number is equal to zero. If it is it multiplies this number by 2 and appends it to a new array. If the index modulus by 2 is not 0, the number is added to another array. These two arrays are then added together. Now, a variable is created by taking the number 10 and subtracting the total of the two added arrays together from it and calculating the modulus by 10. If the result is equal to zero, the checksum is valid and zero will be appended to the string. If the outcome is not equal to zero, the new variable is subtracted from ten, and the resulting number is appended to the string giving us a valid checksum equal to 0 and the number is printed out.

If the input is greater than 12, the number will instead be validated, and should it be between 14 and 19 characters it will call the main checksum function called checksum2.



As we see here, this checksum is a little less complicated. The checksum is the exact same, however, as we are not trying to generate a required final digit to validate and are instead simply validating if the input results in a valid checksum, the result of two arrays is simply modulus by 10. If the result is 0, we have a valid checksum.

Lastly, we have the generate function. This achieved by generating a random string of digits, whereby the leading digits are selected to match the chosen vendor. This ensures that the card is both a valid card number, belonging to the appropriate card vendor as per the user’s selections. Finally, the checksum function is called, and an attempt to validate the number is carried out. Should the card number fail to validate, the process will loop until a valid number is generated.

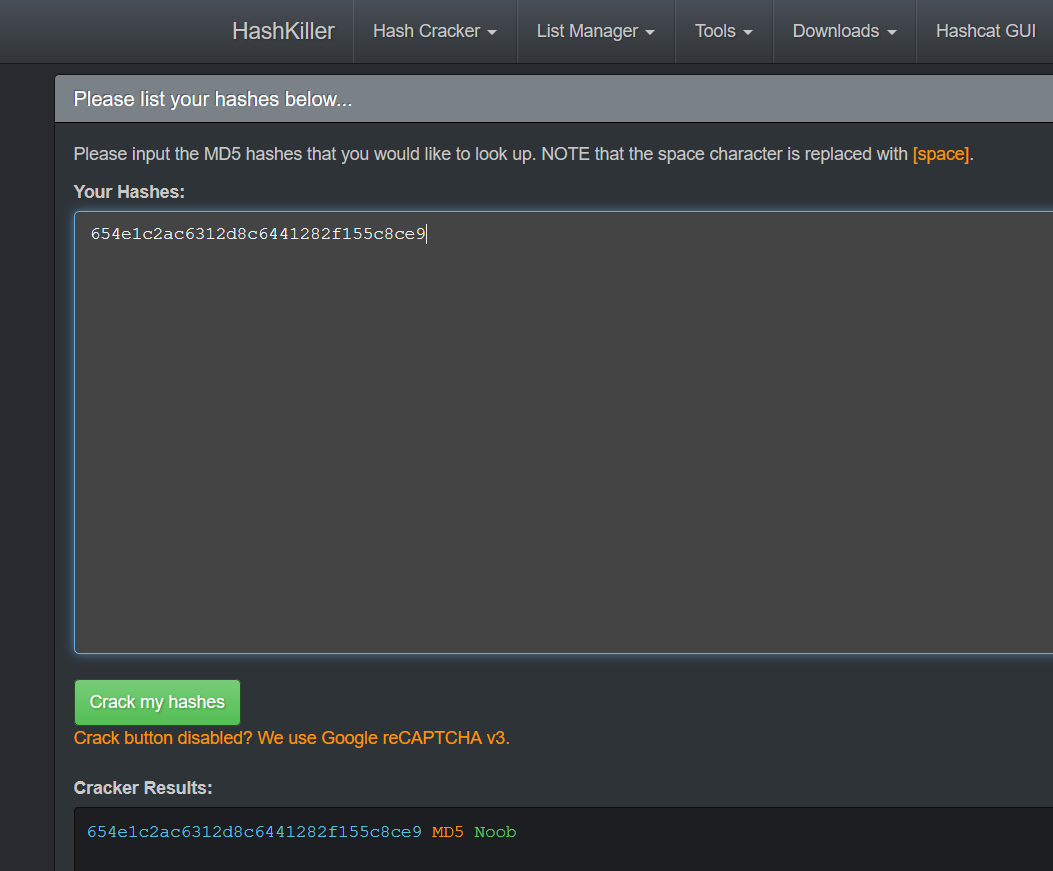


Here we see the function that matches the user selection of an American Express Card. The function generates a random number between 34 and 37, and then a random number between 000000000000 and 99999999999999 which is appended to the previous number. This new number is then validated by the checksum2 function until a valid checksum is generated. We then have a valid card number, belonging to a valid card vendor with a valid checksum.  
  
There are more mini functions that tie all of these functions together, but to cover them would I feel exceed the allotted word count and scope of this report. However, I am confident that I can explain the whole program should it be required if it is called for scrutiny during our lab final demo. I greatly apricated the challenge this lab offered. It was not easy to develop this program in a language I have almost no experience of, but I found it to be highly enjoyable.

# 3: Simple Blockchain

For this lab we are tasked with discovering the challenge hash for a blockchain hash output. In order to achieve this, we are given some important pieces of information. We are told that the user has registered with the user name of ‘nOOB’ and that hash chain seed for this user name is: 654e1c2ac6312d8c6441282f155c8ce9.

A web search against rainbow tables for this hash tells us the hash in plaintext is ‘Noob”.



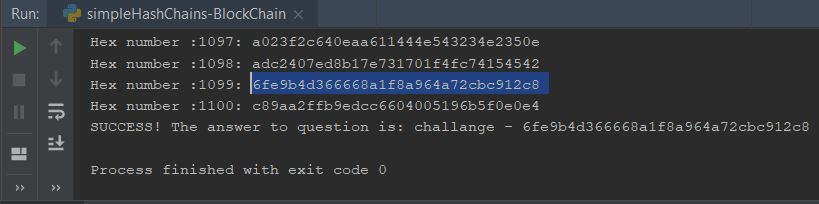
This tells us that the process of this blockchain is simply to invert the username and hash it repeatedly. We do not however know many times it is hashed.

We are then asked to find out how to authenticate as the user ‘ESSC’ using only the hash c89aa2ffb9eedcc6604005196b5f0e0e4. In essence, we must find the hash that when hashed matches the given hash seen above.

This is a simple enough challenge. Knowing the username must be inverted we take the username ‘ecsc’ and hash it until it outputs the hash provided, and then go back 1 hash, and this gives us the hash that when entered will authenticate us as the user ‘ECSC’.

#Author: James Finglas, Technology University Dublin, B00094138  
import hashlib  
from time import sleep  
  
hashSeed = "c89aa2ffb9edcc6604005196b5f0e0e4" #This will be searched online for the inverse of the seed.  
  
some\_string = b"ecsc" #this will be the starting point of the new seed  
  
hash = hashlib.md5(some\_string) #hash some\_string  
  
count = 1  
  
while hash.hexdigest() != hashSeed: #loop hash into rehash untill hash matches hashseed  
 hash = hashlib.md5(hash.hexdigest().encode('utf-8')) #rehash hash  
 print("Hex number :" + str(count) + ": " + hash.hexdigest() + " ") #print out new hash with count number  
 count += 1 #add 1 to count  
 #sleep(0.02) #add 0.02 of a sec wait time between calculatons  
  
if hash.hexdigest() == hashSeed:  
 print("SUCCESS! The answer to question is: challange - 6fe9b4d366668a1f8a964a72cbc912c8") # print this if you succeed in finding the challange  
exit(0)

Here we see my code implemented. And below we can see the output.



# 4: RSA Challenges

### 4.1: RSA Challenge 1

Our first challenge was to Crack an RSA decryption using the provided variables. This task was not one that wold be graded but was simply designed to ease us into cracking RSA and help us understand how RSA works. Our Lecturer, Mark Cummins provided us with a program to fulfil this require which can be found in this documents appendix**(a).**

What can discern from this program is the basic functionality of encrypting and decrypting RSA when we have the required variables. First, we use the string2Int function to convert out input to from a string, to an integer. In this case, we use hardcoded variables as n, e, d, p, and q which are the variables required by the RSA mathematical formula.  
  
We then declare as cipher text is power of the user input (m = message) multiplied by e multiplied by n. This gives us our encrypted message.  
  
To decrypt the message, we simply reverse the process by declaring the decrypted text is the power of (ciphertext multiplied by modulus n), and we convert this int, back to string using the provided int2String.  
  
Here we see the output of the program in action:

"D:\Storage\USB Backup\Year 3\Secure Communications\Secure Communications Labs B00094138\securecomms\venv\Scripts\python.exe" "D:/Storage/USB Backup/Year 3/Secure Communications/Secure Communications Labs B00094138/securecomms/RsaChallenges\_ForSubmission/marksRsaProgram.py"

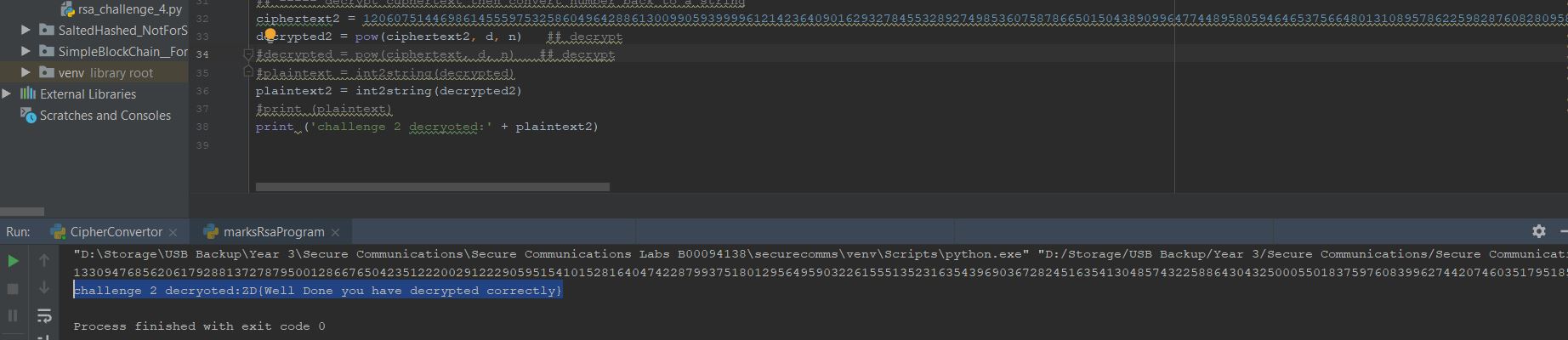
13309476856206179288137278795001286676504235122200291222905951541015281640474228799375180129564959032261555135231635439690367282451635413048574322588643043250005501837597608399627442074603517951858976430767446724730937928672932493206869274420288717036712376949408229648116702610597844919828482630797157003777363091998366855062763360538948110895070725322039940644906900772757193759215740687066380017485804644723367158972689710477927318380335919282326398046586751715463059075476044138690978986063001880735783893361380726584661054926968590764176030209214513123458853087059980258593405395678238799024217478961749328706800

RSA isn't really that hard

Process finished with exit code 0

### 4.2: RSA Challenge 2

The second challenge was again not to be graded but was designed to teach us how to reverse the process of encryption. As I have already explained the process of decryption using the provided I dived into this one right away as the solution was obvious to me.  
  
I simply altered the program provided, commenting out the decryption variables provided, and creating a second ciphertext variable called ciphertext2, into which I fed the provided sequence to be decrypted as seen below. I then duplicated the line of code to decrypt the text using this new variable. And sure enough the sequence decrypted perfectly.

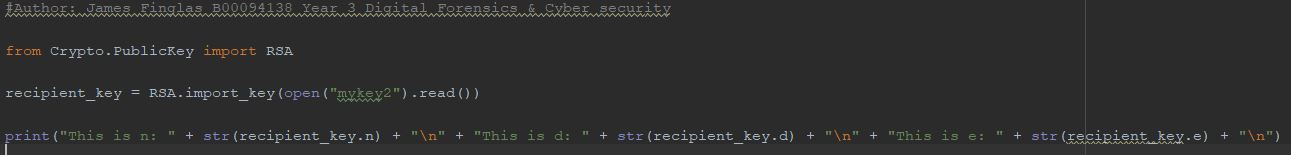


Decrypted text: ZD{Well Done you have decrypted correctly}.

### 4.3 RSA Challenge 3

Challenge three caused me a few headaches at first to wrap my brain around. I am still finding my feet with Python, having never used the language prior to this module and I find the API to often quite poorly documented and not very well explained.  
  
However, thanks to excellent help from my lecturer Mark Cummins I soon began to make progress once I stopped overthinking things and resorted to examining the examples provided in the API in conjunction with function documentation.

The task for challenge 3 was establish the values for n, d and e from a provided sample RSA private key. I came up with a very simple program to do this. Which makes use of the RSA.import\_key() function create a key object by importing the provided key file data from ‘mkey2.txt’, and the break it down to its constituent pieces of data. I then added a line to print out the required data elements using str(recipient\_key.n), str(recipient\_key.d) str(recipient\_key.e). As seen blelow:



And here, we see the data output:

"D:\Storage\USB Backup\Year 3\Secure Communications\Secure Communications Labs B00094138\securecomms\venv\Scripts\python.exe" "D:/Storage/USB Backup/Year 3/Secure Communications/Secure Communications Labs B00094138/securecomms/RsaChallenges\_ForSubmission/rsa\_challange\_3.py"

This is n: 25804750360904248224329381618104859031736073222395248860499133051128513648896523325052435992237099000924849024564826844234697029915822571039994777064514824635243492963301478168480948385273501322600327794162141312212647635059765199895820389754402745012084674140342116327728380193544092902468871988407101113210818201750378837933282937750636399588218292079918162950294282004250547116764896324653457065316101365613347914317959505469586731762453963212022535424803572822031093301253062385003468990260150084465026582297271243286096414015105084255301102689693963566336024015793743350044649378818363190804537712326729609503119

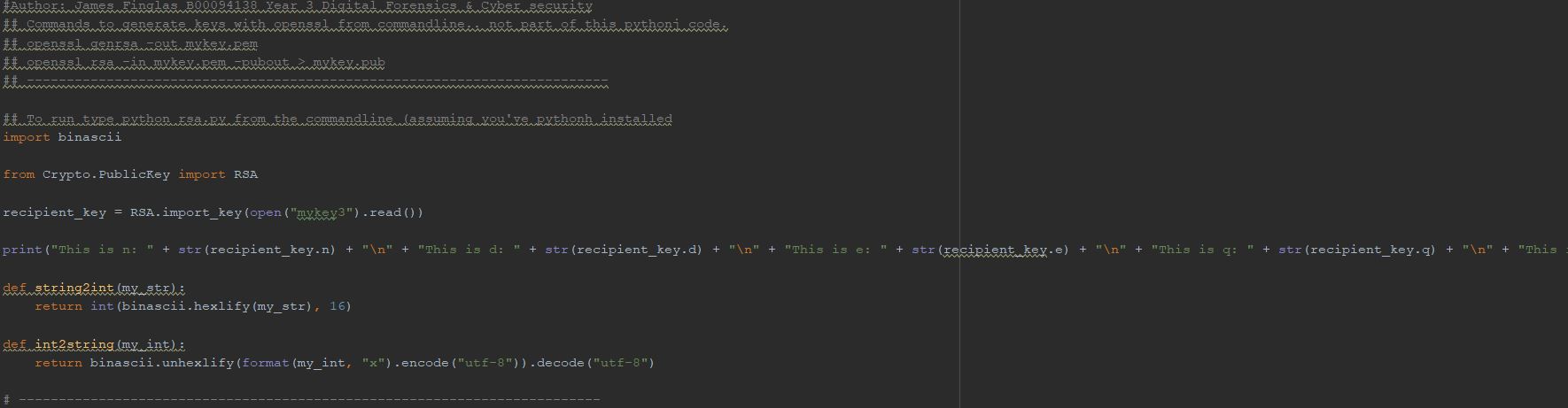
This is d: 25564173244610971210351548452585197331194535758994673657884351015114666175874020376311922413781312689534869581593541780807512387877301314484073132668426835287971970400465565582293880630212968962130535153634911375813942663745522611172864711921423815900248860235223648392420965446176110834882493594752879241898541637225074107295433979711279703743206408857411263301020537809364614125840424406332180800207630227259634428188684178985515460114123914453650465520564263249152086786817100938246123146166512963816583276055667323176816331368813221646162976565638562106256552057858358804329568634212548081342870432443412497223185

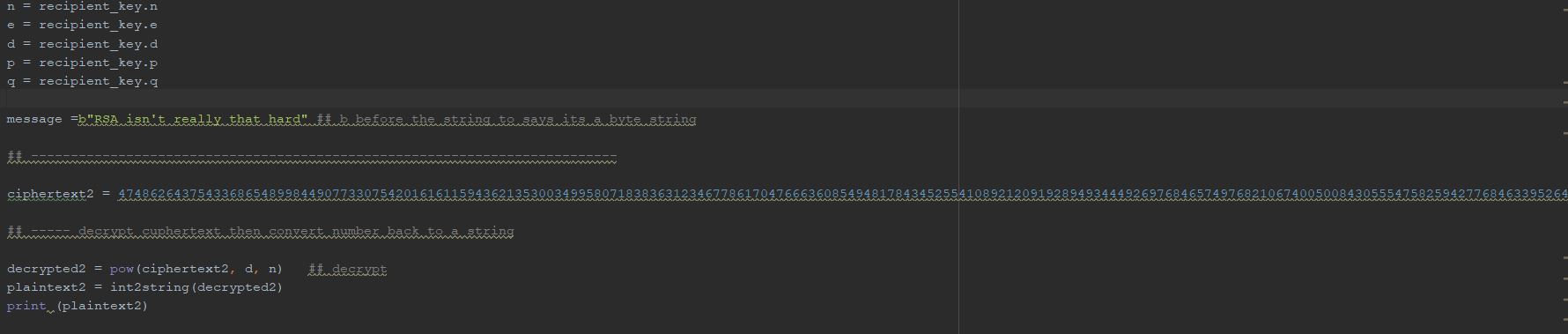
This is e: 65537

Process finished with exit code 0

### 4.4: RSA Challenge 4

The next challenge combined both programs. Our task is to decrypt an RSA private key. Having been provided with a private key, and some cipher text I set out to modify both previous programs while combining them. First I re wrote my code which extracts the data segments from the RSA key, and this time assigned the variables to take the place of those used in the original provided program.  
  
I then input the cipher text and ran the program, this time decrypting using the variables extracted from the provided private RSA key file which gave me the correct output as seen below.





Decrypted text: ZD{OK time to move onto some harder stuff}

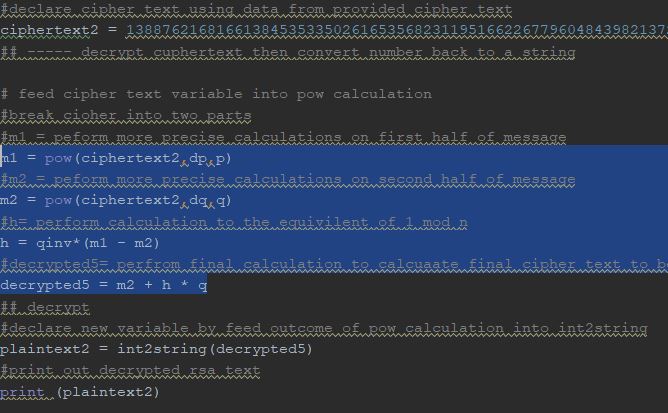
### 4.5 RSA challenge 5

Ra challenge delved into a new area. We are provided with a new set of unfamiliar variables, and out challenge also contains a which direct us this webpage(1). This code formed the basis of my program’s calculations.

The webpages describe the use of the Chinese Remainder algorithm. The algorithm is as follows:

m1 = pow(ciphertext2,dp,p)  
#m2 = peform more precise calculations on second half of message  
m2 = pow(ciphertext2,dq,q)  
#h= perform calculation to the equivilent of 1 mod n  
h = qinv\*(m1 - m2)  
#decrypted5= perfrom final calculation to calcuaate final cipher text to be converted back to string  
decrypted5 = m2 + h \* q

The message is broken into two parts m1 and m2, with m1 being equal to the power of the ciphertext , in binary, multiplied by dp modulus p, and m2 bieing the power of the cipher text multiplied by dq modulus q. Next, we perform a caluculation which gives us the equivalent of 1 mod n, before finally, declaring the final decrypted cipher text by decalring it as m2 + h multiplied by q.

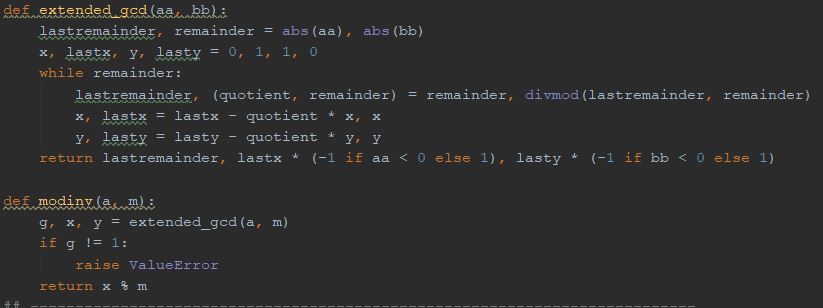


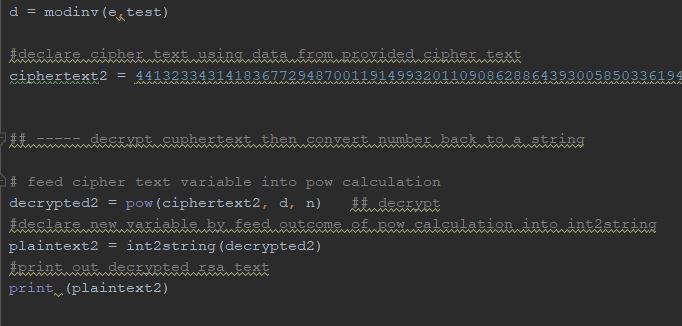
Next, we simply convert this int back to a string as we have done in previous challenges. With the decrypted text being: Those extra private key values are meant to make it easier? This was not an easy challenge to understand as I approached it attempting to understand the math, when in fact I merely needed to understand the new method, which serves to speed up the process by allowing us to use what amounts to short hand variables to quickly calculate, resulting in faster processing of RSA encryption and decryption.

### 4.6 RSA challenge 6

Challenge 6 once again pushed us to explore new methods of extracting data using unknown methods.

We are presented with a cipher text sequence and the variables e, p and q. We are asked to help determine what d is, as this will be needed to decrypt the cipher text.  
  
Returning to our earlier lecture notes to revise, I was remind that: d = modinv(e(p-1)(q-1)). Now seeing as I again don’t fully understand the math here and having watched me struggle for some time much to his amusement I am sure, I was thankfully provided with a clue by my lecturer. He recommended I investigate the rosetta code website (2). Having explored this website, and searched for a modinv() function I discovered one for python.  
  
However, when I attempted to implement it I ran into an issue. The method required to calculate modinv() take sin only two variables. As we have seen above, there are three variables used to calculate d. therefore I simply declared a new variable, test, as being equal to (p-1)\*(q-1). This allowed me to simple feed in e and new new variable which calculated the mod inverse. This means that d is now = modinv(e,test) as seen below.





Now that we have calculate d, we can simply enter the variables and decrypt as we have done previously, with our decrypted text being: You are doing very well, you must be starting to understand RSA by now!  
  
And sure enough, as usual, Marks words prove correct, I am indeed beginning to truly understand RSA.

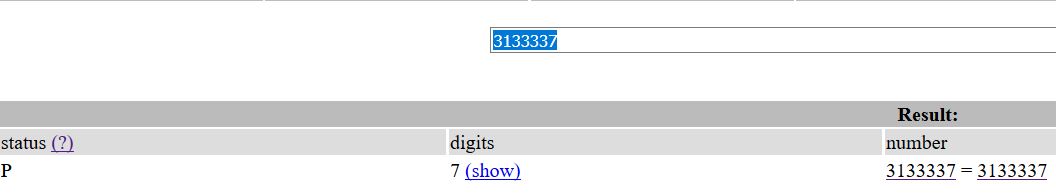
### 4.7 RSA Challenge 7

Challenge 7 introduces another twist. This time we must attempt to extract p and q from the provided data. We are provided with another hint for this challenge, the hint being: FactorDB is your friend!

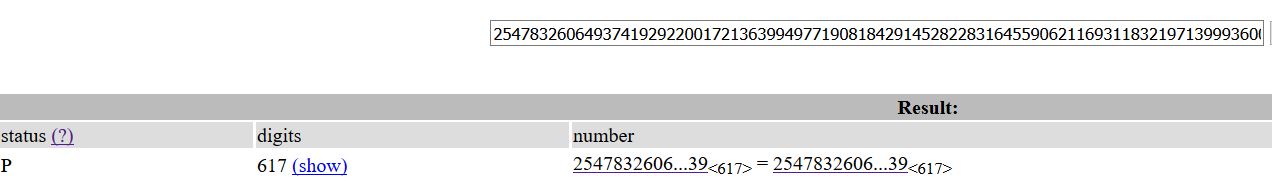
We are provided with a file containing the cipher text, and the variables e and n. Returning to lecture note again I am reminded that q and p are factors of n. I must therefor factorize n to retrieve the data points.

I tumbled down a rabbit hole here. Following my instincts, I searched for factorDB python, and sure enough I found a package and I was

able to install the package and import the function factorDB(). However, I could not find adequate information within the python API to implement a manual factorization. Having consulted briefly with my lecturer and fellow students I was pointed to a website, also called FactorDB, which did exactly what I was trying to do, without any in program implementation nessecary.



Here we see p extracted, and below we can see q extracted. The UI admittedly is not very useable on this website, but the functionality, once you understand how to use it is good.



Now that we have extracted p and q, we can feed them into our very first RSA decryption calculate we used in challenge 1 and get our decrypted cipher which reads: Only 4 more challenges to go!

### 4.8 RSA Challenge 8

RSA challenge 8 was designed to teach us how to manually retrieve the cubic root of large integers. While a simple cubic root function does exit in python it does not work large integers.  
  
I solved this challenge very quickly by retrieving the cubic root via factorDB.com. However, I did not fully understand the problems related to cubic roots in python. It was explained to me that this was a special circumstance where factoring would work but in almost every other case that would not work. While my solution did solve the challenge, it did t0 produce a calculation that would work for other challenges.  
  
So I returned to the drawing board to search for functions that would achieve the desired result. I found the perfect calculation in stack overflow here **(3)**, which enables a coder to calculate the cube, and cubic root of large integers in python.

minprec = 27  
if len(x) > minprec: getcontext().prec = len(x)  
else: getcontext().prec = minprec  
  
x = Decimal(x)  
power = Decimal(1)/Decimal(3)  
  
answer = x\*\*power  
ranswer = answer.quantize(Decimal('1.'), rounding=ROUND\_UP)  
  
diff = x - ranswer\*\*Decimal(3)  
if diff == Decimal(0):  
 print("x is the cubic number of", ranswer)  
 print(" ")  
 print("x has a cubic root of ", answer)  
 print(" ")  
  
ciphertext2 = ranswer  
print(int2string(int(ranswer)))

the method to compute the power is redefined as being equal to the input raised to the power of 1/3. The answer is then rounded up to the nearest decimal value. This gives us quite a precise and quick calculation. The decrypted text is: We always need to watch the size of our message

### 4.9 RSA Challenge 9

RSA challenge 9 was quite difficult. First, we must import the code for modular inverse once again from Rosetta Code as discussed in a previous challenge **(2)**. I cannot pretend to fully understand the codes implementation, except to state it involves the use of the Chinese remainder theorem implemented via the Gauss algorithm. I used this site to help guide me through this process **(4)**.

Here we can see the implementation of the Chinese remainder theorem via the Gauss algorithm:

N1 = n2 \* n3  
N2 = n1 \* n3  
N3 = n1 \* n2  
N = n1 \* n2 \* n3  
d1 = modinv(N1,n1)  
d2 = modinv(N2,n2)  
d3 = modinv(N3,n3)  
c = ((c1\*N1\*d1) + (c2\*N2\*d2) + (c3\*N3\*d3)) % N

Our decrypted text is: Impressive: small\_e\_is\_the\_killer 38247601923468.

### 4.10 RSA Challenge 10

Rsa challenge 10, was for an extremely difficult task as it involved thoroughly understand the method used in the attack type, which is the common modulus attack. This requires us to work out what the s values are.  
  
In order to be able to work out what ‘s1’ and ‘s2’ are we need to understand some basic things at a basic level. For example: if the extended Euclidean algorithm calculation of e1 by e2 is equal to 1 then, there exist some s1,s2 such that eBs1+eCs2=1. This tells us that we are looking for s1 and s2. It further tell us that we can extract these by again taking the ‘egcd’ code from Rosetta Code covered earlier and importing it into python. This will give us s1 and s2, as seen below.

d, s1, s2 = egcd(e1,e2) # get inverse of s1 or s2 if < 0

As my comment states, one or both of these can potential by less than zero, in which case, we must return the inverse of either s1 or s2, or both in order to be able to decrypt the message.  
  
Now we must retrieve the message suing the values s1 and s2 we have extracted.

This is done using this formula:

(e1 \* s1) + (e2 x s2) =1

C1=me, c2 = me2 therefore,

Pow(C1,s1) \* pow(c2,s2) = pow(pow(m,e1)s1) \* pow(pow(m,2)s2)

= (pow(pow(m,e1)s1) \* (pow(pow(m,e2)s2)

As we have seen above the powers on the left-hand side being added to the powers on the right hand side result in 1, so the entire equation becomes pow(m,1). This in turn becomes ’m’, giving us our decrypted message.

if s1 < 0:  
 c1 = modinv(c1, n1)  
 s1 = -s1  
  
if s2 < 0:  
 c2 = modinv(c2, n1)  
 s2 = -s2  
  
message = pow(c1,s1,n1) \* pow(c2,s2,n1) % n1  
  
print(bytes.fromhex(hex(message)[2:]))

As we see above we prepare two if statements in case the numbers s1 or s2 are less then 0 and need to be inverted, when one did in this case.  
  
Finally the message is extracted. I did need a small bit of help in the conversion process, for which I am thankful a fellow student could help out, still being new to python, as I am unsure why the conversion resulted in my being unable to convert the message from into to string as previously done. However, once converted the decryption proved correct. The decrypted text is: Only one more to go.. the force is strong with this one!.

### 4.11 RSA Challenge 11

# References:

1. <https://en.wikipedia.org/wiki/RSA_(cryptosystem)>.
2. <http://www.rosettacode.org/wiki/Modular_inverse#Python>.
3. <https://stackoverflow.com/questions/356090/how-to-compute-the-nth-root-of-a-very-big-integer/637321?fbclid=IwAR1rH3H74lxdStZAZM8rplXHbCmGv4tqi3fdujVBASM7a-GuoIJFUWFWr2I>.
4. https://di-mgt.com.au/crt.html.

# Appendix

1. Program Provided by Mark Cummins:

## Commands to generate keys with openssl from commandline.. not part of this pythonj code.

## openssl genrsa -out mykey.pem

## openssl rsa -in mykey.pem -pubout > mykey.pub

## -------------------------------------------------------------------------

## To run type python rsa.py from the commandline (assuming you've pythonh installed

import binascii

def string2int(my\_str):

return int(binascii.hexlify(my\_str), 16)

def int2string(my\_int):

return binascii.unhexlify(format(my\_int, "x").encode("utf-8")).decode("utf-8")

# -------------------------------------------------------------------------

n = 23516695565660963250242846975094031309572348962900032827958534374248114661507001374384417953124930587796472484525315334716723068326965228898857733318407681656604325744994115789416012096318656034667361976251100005599211469354510367804546831680730445574797161330145320706346512982316782618118878428893337849886890813813050423818145497040676697510093220374542784895778086554812954376689653727580227087363619223145837820593375994747273662064715654881379557354513619477314410917942381406981452545764657853425675230343749326640073923166795823683203941972393206970228647854927797483660176460658959810390117898333516129469397

e = 65537

d = 9587600726595591453426898215169101767863399178169979967502694355028996988583633210586039386751682566723132708455252764519220038491664005843242439790264046968625524201298469258242007220372280857992847470031480553726983707671745159488070659256258857978134570602562717609180653377092666963295822401721181836384326336158085408894694549470434424808812412260714422693522311366681659987060925945689943522825747715934700712908720597323076354591388316712970722935035250113120539406041972135508540472211484760814740089404942374666334486855389174327639061106567747152104666795257954039030591097174242386069752606041990644663125

p = 170436857437540785902894247445629309884819493988198726337160363787266132388801445377172350883259146330710518633323153950488107255453274647690833952071079266615535462115718628529996080297946386916054952930963525522668498855400580516951309863503734146131687670337990358661269686138903141878297721385390421204703

q = 137978932017559751745702136624874154954496829862527332457067512249687998333117572719846957168595861866495967632464915097378576596911015571165340454225721218087595428364080801400548238088288742249145662369868461078198744980520572785232341389134600070345564258064842348774203427257497319140459851255774165194699

message =b"RSA isn't really that hard" ## b before the string to says its a byte string

## --------------------------------------------------------------------------

## ----- convert message to an int then encrypt ------------

m = string2int(message)

ciphertext = pow(m, e, n) ## encrypt

print (ciphertext)

## ----- decrypt cuphertext then convert number back to a string

decrypted = pow(ciphertext, d, n) ## decrypt

plaintext = int2string(decrypted)

print (plaintext)