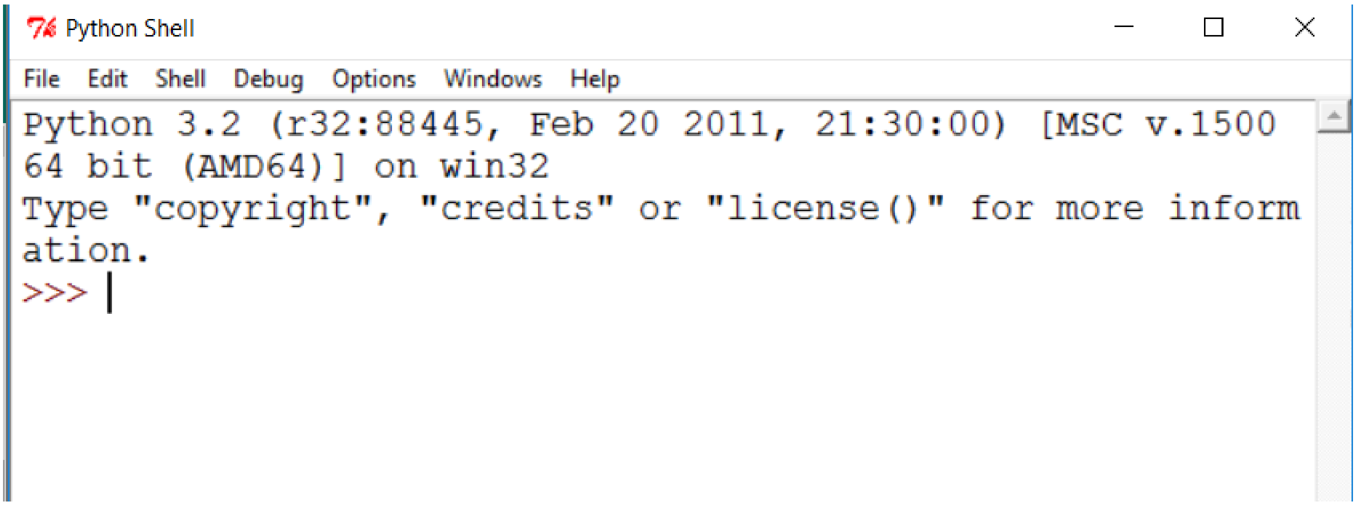
RSA

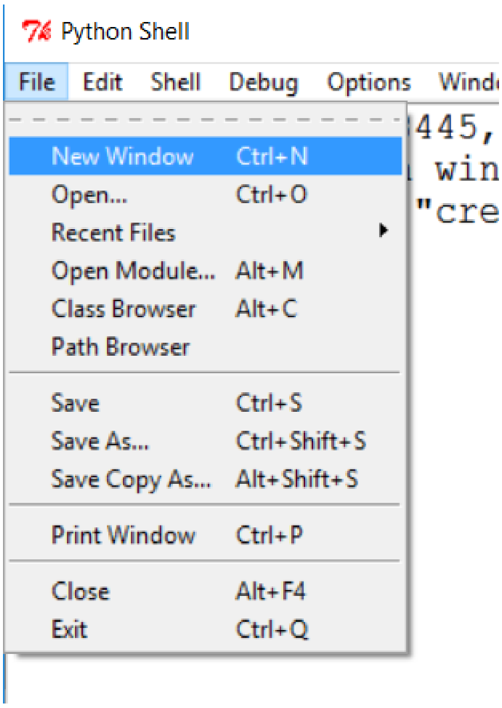
Name:	Class:	Date:

Create a new Python project as follows:

Open IDLE, ideally the newest version.



Click file and create a New Window:



Name this file:

RSA.py

RSA

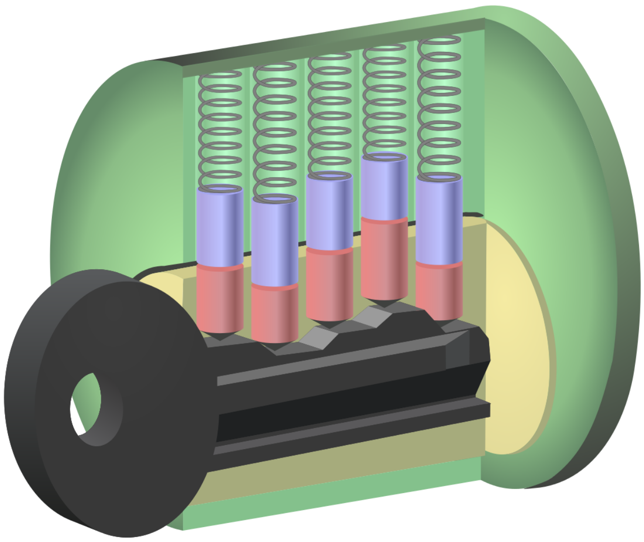
The RSA (Rivest–Shamir–Adleman) Cipher is a kind of asymmetrical cryptography that unlike a symmetrical cipher makes use of 'keys' to encrypt messages.

Keys are generated using three prime numbers to create a public and private key.

The idea is that you give others your public key to 'encrypt' and you have a private key that can only 'decrypt'.

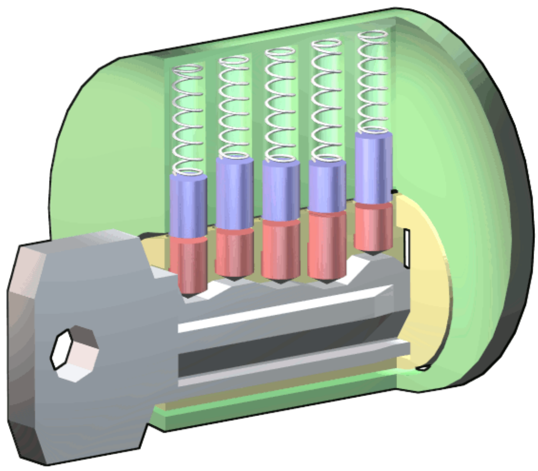
Unlike symmetrical encryption which can easily be cracked with search algorithms, asymmetrical encryption is almost impossible.

If it helps imagine a lock that has two keys.



The public key matches the tumblers position if the lock is open and it will allow it to turn, however once the tumblers move into position the tumblers move into a position that does not match the 'locking' key.

The private key's teeth match the position of the locked tumblers meaning that it can be unlocked.



However once unlocked, the tumblers move into a position that the private keys teeth do not match.

In simple terms, the 'lock' lacks symmetry in the way the tumblers fall for locking and unlocking.

RSA

You will create a basic RSA Cipher that uses a predefined library named rsa-maths.py. This Python code is included with the lesson, you do not need to create this code, but the code in the file looks like this:

rsamaths.py - C:\Users\richard.hunt\OneDrive - Hugh Baird College\Resources\Programmin...

File Edit Format Run Options Windows Help

```python
#Credit: H Mcgowan
#Author: R Hunt
#Reference: Sweigart, A (2013). Hacking Secret Ciphers with Python. Pp. 196-201,
import math

# Cryptomath Module
# http://inventwithpython.com/hacking (BSD Licensed)

def gcd(a, b):
    # Return the GCD of a and b using Euclid's Algorithm
    while a != 0:
        a, b = b % a, a
    return b


def find_mod_inverse(a, m):
    # Returns the modular inverse of a % m, which is
    # the number x such that a*x % m = 1

    if gcd(a, m) != 1:
        return None # no mod inverse if a & m aren't relatively prime

    # Calculate using the Extended Euclidean Algorithm:
    u1, u2, u3 = 1, 0, a
    v1, v2, v3 = 0, 1, m
    while v3 != 0:
        q = u3 // v3 # // is the integer division operator
        v1, v2, v3, u1, u2, u3 = (u1 - q * v1), (u2 - q * v2), (u3 - q * v3), v1
    return u1 % m
```

This code uses maths that we need, but since it is already established code it is best to reuse rather than rewrite.

RSA

We will now look at writing the actual RSA Cipher which will include three functions:

generate_keys

encrypt

decrypt

We will start with generate_keys:

```
import rsamaths

#Preconditions: p, q, e are all Integers that are prime numbers.
#Postconditions: Return a public and private key.
def generate_keys(p, q, e):
    n = p*q                                 #n is used in both keys.
    print("n",n)
    phi_n = (p-1)*(q-1)                     #phi_n, used to generate d later.
    print("phi(n)",phi_n)
    d = rsamaths.find_mod_inverse(e, phi_n) #Uses Euclidean Algorithm.
    print("d", d)
    print("public key", [n,e])
    print("private key", [n,d])
    public_key = n,e                        #Public key is n,e
    private_key = n,d                       #Private key is n,d
    return public_key, private_key          #Returns public and private keys
```

This code will take three prime numbers p, q, e.

From p * q this it will create ‘n’ which will be in both public and private keys.

It will also create phi_n using (p-1) * (q-1) .

From there d is created using the rsamaths library with e, phi_n.

The public key will be [n, e]

The private key will be [n, d]

RSA

Now we have a way to generate keys we can move onto actually encrypting the message.

In many ways this encryption algorithm is not too different to one for symmetrical encryption, it takes a character, turns it into a number, applies a shift 'public_key[1]' and applies modulus 'public_key[0].

```
#Preconditions: Takes a phrase as string, and the public key as list of two elements.
#Postconditions: Returns and encrypted phrase.
def encrypt(phrase, public_key):
    enc_phrase = ""                                     #Empty string to hold the encrypted phrase.
    m = 0                                               #To hold the integer value fo a character.
    for i in phrase:
        m = pow(ord(i), public_key[1]) % public_key[0]  # (i^public_key[1]) MOD public_key[0]
        print(m)
        enc_phrase = enc_phrase + chr(m)                #Adds the character m to the enc_phrase.
        m = 0                                           #Reset m
    return enc_phrase                                   #Return encrypted phrase.
```

Now all we need is a decrypt function to undo the message.

As with a Caesar Cipher, the algorithm is almost identical, the only clue I will give is that you need to change the key!

Once that is done, write the following code to activate the functions and provide some prime numbers.

```
p_val = int(input("p:",))
q_val = int(input("q:",))
e_val = int(input("e:",))
pub_key, priv_key = generate_keys(p_val, q_val, e_val)
result = encrypt("1 love is all around", pub_key)
print(result)
print(decrypt(result, priv_key))
```

Here is some p, q, e values that will work:

#Valid p, q, e

#23, 37, 85 #7, 31, 7

RSA

Name: | Class: | Date:

Final note:

You will find that randomly putting prime numbers in will not always produce valid results.

This is because this version of RSA lacks many of the validation methods of the full cipher.

Some combinations of p, q, e will not allow for encryption and decryption to work.

This link is a website that has an RSA that will generate valid p, q, e values.

https://asecuritysite.com/encryption/rsa

Credits and references:

Yufei Tao, RSA Cryptosystem https://www.cse.cuhk.edu.hk/~taoyf/course/bmeg3120/notes/rsa.pdf

H McGowan—Hugh Baird College A student of mine that was very helpful solving an issue with the rsamaths.py.