###MYSQL-Python-User\_Interface###:

import Euler\_algor

import Config\_handler

import os

import csv

import mysql.connector

##MYSQL-Connection##

class mysql\_connection():

def \_\_init\_\_(self, user, password, host, database):

self.user = user

self.password = password

self.host = host

self.database = database

self.connection\_name = ''

self.connection\_details = {}

self.connections = {}

self.uncommited\_changes = True

self.turn\_off\_autocommit = ("SET autocommit = 0")

self.create\_table\_statement = ("CREATE TABLE {} ({})")

self.delete\_table\_statement = ("DROP TABLE {}")

self.create\_database\_statement = ("CREATE DATABASE {}")

self.delete\_database\_statement = ("DROP DATABASE {}")

self.show\_databases\_statement = ("SHOW DATABASES")

self.show\_tables\_in\_database\_statement = ("SHOW TABLES IN {}")

self.show\_columns\_in\_table\_statement = ("SHOW COLUMNS IN {}")

self.select\_all\_table\_data = ("SELECT \* FROM {}")

self.database\_blacklist = ['information\_schema', 'mysql', 'performance\_schema', 'sakila', 'sys']

def pass\_back\_cursor(self, connection\_name):

try:

connector = self.connections[connection\_name]

curA = connector.cursor(buffered=True)

except:

return False

else:

return curA

def get\_database\_names(self, connection\_name):

try:

databases = list(self.connection\_details.keys())

except:

return False

else:

return databases

def get\_table\_names(self, database\_name, connection\_name):

try:

table\_names = list(self.connection\_details[database\_name].keys())

except:

return False

else:

return table\_names

def get\_table\_attributes(self, table\_name, connection\_name):

try:

connector = self.connections[connection\_name]

#create cursor

curA = connector.cursor(buffered=True)

select\_table\_attribute = self.show\_columns\_in\_table\_statement.format(table\_name)

curA.execute(select\_table\_attribute)

except:

return False

else:

data = curA.fetchall()

attributes = [attribute[0] for attribute in data]

return attributes

def get\_table\_data(self, table\_name, connection\_name):

try:

connector = self.connections[connection\_name]

#create cursor:

curA = connector.cursor(buffered=True)

select\_table\_data\_statement = self.select\_all\_table\_data.format(table\_name)

curA.execute(select\_table\_data\_statement)

except:

return False

else:

data = curA.fetchall()

return data

def refresh\_connection(self, connection\_name):

try:

connector = self.connections[connection\_name]

connector.config(database = self.database)

#This will rollback any uncommited changes

connector.reconnect()

self.get\_connection\_details(connection\_name)

except:

return False

else:

return True

def commit\_changes(self, connection\_name):

try:

connector = self.connections[connection\_name]

connector.commit()

except:

return False

else:

return True

def rollback\_changes(self, connection\_name):

try:

connector = self.connections[connection\_name]

connector.rollback()

except:

return False

else:

return True

def create\_new\_database(self, database\_name, connection\_name):

try:

connector = self.connections[connection\_name]

#create cursor:

curA = connector.cursor(buffered=True)

#Turn off autcommit:

curA.autocommit = False

database\_creation\_statement = self.create\_database\_statement.format(database\_name)

curA.execute(database\_creation\_statement)

self.connection\_details.update({database\_name:{}})

except:

return False

else:

return True

def delete\_database(self, database\_name, connection\_name):

try:

#Choose database to switch to:

database\_name2 = self.connection\_details.keys()[0]

if database\_name == self.connection\_details.keys()[0]:

database\_name2 = self.connection\_details.keys()[1]

self.change\_database(database\_name2, self.connection\_name)

connector = self.connections[connection\_name]

#create cursor:

curA = connector.cursor(buffered=True)

#Turn off autcommit:

curA.autocommit = False

database\_deletion\_statement = self.delete\_database\_statement.format(database\_name)

curA.execute(database\_deletion\_statement)

self.connection\_details.pop(database\_name)

except:

return False

else:

return True

def create\_new\_table(self, table\_name, attributes, attribute\_types, connection\_name):

if len(attributes) != len(attribute\_types):

raise Exception("attibutes and attribute\_types, are not the same length")

else:

try:

attributes\_types\_string = ''

for i in range(len(attributes)):

attributes\_types\_string += attributes[i] + ' ' + attribute\_types[i]

if i != 0 or len(attributes) -1:

attributes\_types\_string += ','

print(attributes\_types\_string)

connector = self.connections[connection\_name]

#create cursor:

curA = connector.cursor(buffered=True)

#Turn off autcommit:

curA.autocommit = False

table\_creation\_statement = self.create\_table\_statement.format(table\_name, attributes\_types\_string)

curA.execute(table\_creation\_statement)

self.connection\_details[self.database].update({table\_name:attributes})

except:

return False

else:

return True

return False

def delete\_table(self, table\_name, connection\_name):

try:

connector = self.connections[connection\_name]

#create cursor:

curA = connector.cursor(buffered=True)

#Turn off autcommit:

curA.autocommit = False

table\_deletion\_statement = self.delete\_table\_statement.format(table\_name)

print(table\_deletion\_statement)

curA.execute(table\_deletion\_statement)

print('hi')

self.connection\_details[self.database].pop(table\_name)

except:

return False

else:

return True

def get\_connection\_details(self, connection\_name):

try:

connection\_details\_dict = {}

#Get Databases:

connector = self.connections[connection\_name]

curA = connector.cursor(buffered=True)

curB = connector.cursor(buffered=True)

curC = connector.cursor(buffered=True)

curA.execute(self.show\_databases\_statement)

for database in curA:

if database[0] not in self.database\_blacklist:

database\_tables\_query = self.show\_tables\_in\_database\_statement.format(database[0])

connection\_details\_dict.update({database[0]: {}})

curB.execute(database\_tables\_query)

for table in curB:

columns\_table\_query = self.show\_columns\_in\_table\_statement.format(database[0]+'.'+table[0])

curC.execute(columns\_table\_query)

attribute\_array = [attribute[0] for attribute in curC]

connection\_details\_dict[database[0]].update({table[0]: attribute\_array})

except:

if err.errno == errorcode.ER\_ACCESS\_DENIED\_ERROR:

print("Something is wrong with your user name or password")

elif err.errno == errorcode.ER\_BAD\_DB\_ERROR:

print("Database does not exist")

else:

print(err)

return False

else:

self.connection\_details = connection\_details\_dict

return True

def setUp\_connection(self, connection\_name):

try:

if connection\_name in self.connections.keys():

connection\_name = connection\_name + '#2'

connector = mysql.connector.connect(

user = self.user,

password = self.password,

host = self.host,

database = self.database)

self.connections.update({connection\_name:connector})

except:

if err.errno == errorcode.ER\_ACCESS\_DENIED\_ERROR:

print("Something is wrong with your user name or password")

elif err.errno == errorcode.ER\_BAD\_DB\_ERROR:

print("Database does not exist")

else:

print(err)

return False

else:

self.connection\_name = connection\_name

return True

def change\_database(self, database\_name, connection\_name):

if database\_name == self.database:

return True, 1

try:

connector = self.connections[connection\_name]

connector.config(database = database\_name)

#This will rollback any uncommited changes

connector.reconnect()

self.database = database\_name

except:

if err.errno == errorcode.ER\_BAD\_DB\_ERROR:

print("Database does not exist")

else:

print(err)

return False, 1

else:

return True, 2

def test\_connection(self, connection\_name):

try:

connector = mysql.connector.connect(

user = self.user,

password = self.password,

host = self.host,

database = self.database)

except:

return False

else:

return True

def terminate\_connection(self, connection\_name):

if connection\_name not in self.connections.keys():

return False

connector = self.connections[connection\_name]

connector.close()

self.connections.pop(connection\_name)

return True

def terminate\_all\_connections(self):

for key in self.connections.keys():

connection = self.connections[key]

connection.close()

return True

def turn\_off\_clients\_autocommit(self):

config\_obj = Config\_handler.config\_obj(data\_path, filename, curfile)

config\_obj.find\_mysql\_config\_file()

status\_code = config\_obj.set\_auto\_commit\_zero()

return status\_code[-1]

"""

#Test1:

user = 'root'

password = 'Joshua100x'

host = '127.0.0.1'

database = 'world'

connection\_name = 'connection1'

mysql\_connector = mysql\_connection(user, password, host, database)

mysql\_connector.setUp\_connection(connection\_name)

mysql\_connector.get\_connection\_details(connection\_name)

for connection in mysql\_connector.connection\_details.items():

print('\n', connection,'\n')

"""

#Test2:

table\_name = 'clients'

user = 'root'

password = 'Joshua100x'

host = '127.0.0.1'

database = 'sql\_invoicing'

connection\_name = 'connection1'

mysql\_connector = mysql\_connection(user, password, host, database)

mysql\_connector.setUp\_connection(connection\_name)

mysql\_connector.get\_connection\_details(connection\_name)

#Test table attribues and data

table\_connections = mysql\_connector.get\_table\_data(table\_name, connection\_name)

table\_attributes = mysql\_connector.get\_table\_attributes(table\_name, connection\_name)

table\_names = mysql\_connector.get\_table\_attributes(table\_name, connection\_name)

print(table\_names)

print(table\_attributes)

print(table\_connections)

#Test table and database creation:

database\_names = mysql\_connector.get\_database\_names(connection\_name)

print(database\_names)

#Test if the connector can add a new database (and if it apears in connection\_details) and if it only existis in the session like its supposed to:

database\_names = mysql\_connector.get\_database\_names(connection\_name)

print(database\_names)

mysql\_connector.create\_new\_database('database1', connection\_name)

database\_names = mysql\_connector.get\_database\_names(connection\_name)

print(database\_names)

mysql\_connector.refresh\_connection(connection\_name)

database\_names = mysql\_connector.get\_database\_names(connection\_name)

print(database\_names)

#Tets if the connector can add a new table (and if it apears in connection\_details) and if it only existis in the session like its supposed to:

print('hi')

table\_name = 'customers'

attributes = ['name']

attribute\_types = ['VARCHAR(25)']

table\_names = mysql\_connector.get\_table\_names(database, connection\_name)

print(table\_names)

is\_created = mysql\_connector.create\_new\_table(table\_name, attributes, attribute\_types, connection\_name)

print(is\_created)

table\_names = mysql\_connector.get\_table\_names(database, connection\_name)

print(table\_names)

mysql\_connector.terminate\_all\_connections()

"""

#Test to see if the tables are on autocommit:

#Result: They are :(:

mysql\_connector.refresh\_connection(connection\_name)

table\_names = mysql\_connector.get\_table\_names(database, connection\_name)

print(table\_names)

#Delete table:

table\_name = 'customers'

table\_names = mysql\_connector.get\_table\_names(database, connection\_name)

print(table\_names)

is\_deleted = mysql\_connector.delete\_table(table\_name, connection\_name)

print(is\_deleted)

table\_names = mysql\_connector.get\_table\_names(database, connection\_name)

print(table\_names)

"""

###MYSQL-Python-User\_Interface###:

##MYSQL-Connection-SQL-Statement##:

#The WHERE condition will be repeated in every single statement and therefore the complining function for it can be a seperate function:

SELECTION\_statement\_str\_WHERE = "WHERE{}"

def compile\_where\_statement(where\_attributes, where\_condition, where\_values, where\_and\_or):

complied\_conditions = ''

min\_attribute = len(where\_arrays\_array)

for i in range(min\_attribute):

where\_attributes\_array = where\_attributes[i].split(' ')

attributes\_length = len(where\_attributes\_array)

for j in range(attributes\_length - 1):

cur\_condition += where\_attributes\_array[j], where\_and\_or[j]

cur\_condition = where\_attributes\_array[attributes\_length]

if i == min\_attribute - 1:

cur\_condition += ' ' + where\_condition[i], where\_values[i], where\_and\_or[attributes\_length]

else:

cur\_condition += ' ' + where\_condition[i], where\_values[i]

compiled\_conditions + cur\_condition

#Bring where\_and\_or up to where\_attributes:

where\_and\_or = where\_and\_or[attributes\_length + 1:]

WHERE\_statement = SELECTION\_statement\_str\_WHERE.format(compiled\_conditions)

return WHERE\_statement

#SELECTION statement#:

#Notes on writing the WHERE statement and ORDER BY statement#:

#When creaing the WHERE statement of the query, if there are two or more attributes in the form:

# - (WHERE att1 AND att2 NOT NULL)

#att1 and att2 will be read into 1 slot in the array with a special charatcer '|' between them to denote the need for an

#'AND' or 'OR' statement

#Whereas when you have a statement like this:

# - (WHERE att1 = 1 AND att2 = NONE)

#they will both be writen into sperate parts of the array

#Conditions and attributes for WHERE and ORDER BY will be written into a string in the order they are in inside of the array

SELECTION\_statement\_str\_SELECT\_FROM = """SELECT {} FROM"""

SELECTION\_statement\_str\_ORDER\_BY = "ORDER BY {} {}"

def create\_SELECT\_statement(is\_all, attributes, table, is\_where, where\_attributes, where\_condition, where\_values, where\_and\_or, is\_order, order\_attributes, order\_desc\_asc):

#Compile SELECT\_FROM section:

SELECT\_FROM\_statement = SELECTION\_statement\_str\_SELECT\_FROM.format(\*attributes)

SELECT\_statement += SELECT\_FROM\_statement

#Compile WHERE section:

if is\_where:

WHERE\_statement = compile\_where\_statement(where\_attributes, where\_condition, where\_values, where\_and\_or)

SELECT\_statement += ' ' + WHERE\_statement

#Compile ORDER BY section:

if is\_order:

ORDER\_BY\_statement = ''

ORDER\_BY\_array = []

for i in range(len(order\_attributes)):

cur\_order = order\_desc\_asc[i]

if not cur\_order:

cur\_order = 'ASC'

compiled\_order = order\_attributes, cur\_order

ORDER\_BY\_array.append(compiled\_order)

ORDER\_BY\_statement = SELECTION\_statement\_str\_ORDER\_BY.format(\*ORDER\_BY\_array)

SELECT\_statement += ORDER\_BY\_statement

return SELECT\_statement

#INSERTION statement#:

INSERTION\_statement\_str\_attributes\_table = "INSERT INTO {} ({})"

INSERTION\_statement\_str\_values = "VALUES {}"

def create\_INSERT\_statement(attributes, table, values):

#Compile attributes and table section:

TABLES\_COLUMNS\_statement = INSERTION\_statement\_str\_attributes\_tables.format(table, attributes)

#Compile values section:

VALUES\_statement = INSERTION\_statement\_str\_values.format(values)

INSERT\_statement = TABLE\_COLUMNS\_statement, VALUES\_statement

return INSERT\_statement

#UPDATE statement#:

UPDATE\_statement\_str\_attributes\_table = "UPDATE {} SET {}"

UPDATE\_statement\_str\_WHERE = "WHERE {}"

def create\_UPDATE\_statement(table, attributes, values, is\_where, where\_attributes, where\_condition, where\_values, where\_and\_or):

UPDATE\_statement = ''

#Compile attributes and table section:

TABLES\_COLUMNS\_statement = UPDATE\_statement\_str\_attributes\_table.format(table, attributes)

UPDATE\_statement += TABLES\_COLUMNS\_statement

#Compile WHERE section:

if is\_where:

WHERE\_statement = compile\_where\_statement(where\_attributes, where\_condition, where\_values, where\_and\_or)

UPDATE\_statement += ' ' + WHERE\_statement

return UPDATE\_statement

#DELETE statement#:

DELETE\_statement\_str\_table = "DELETE FROM {}"

DELETE\_statement\_str\_WHERE = "WHERE {}"

def create\_DELETE\_statement(table, is\_where, where\_attributes, where\_condition, where\_values, where\_and\_or):

DELETE\_statement = ''

#Compile table section:

TABLE\_statement = DELETE\_statement\_str\_table.format(table)

DELETE\_statement += TABLE\_statement

#Compile WHERE section:

if is\_where:

WHERE\_statement = compile\_where\_statement(where\_attributes, where\_condition, where\_values, where\_and\_or)

DELETE\_statement += WHERE\_statement

return DELETE\_statement

###Python-MYSQL-User\_Interface###:

from configparser import ConfigParser

import os.path

import shutil

data\_path = '\\Data\\'

filename = 'Config.ini'

curfile = str(\_\_file\_\_)

##Config\_handler##:

class config\_obj:

def \_\_init\_\_(self, data\_path, filename, curfile):

self.special\_characters = [':',';','.',',','[',']','{','}','/','?','|', '"\"', '"', "'", '=' ,'+', '#', '~', '@', '\*', '&', '^', '%', '$,' ,'£', '!']

self.data\_path = data\_path

self.filename = filename

self.curfile = curfile

self.config\_filepath = ''

self.mysql\_config\_default\_path = "C:\ProgramData\MySQL\MySQL Server 8.0\my.ini"

self.mysql\_config\_file = 'my.ini'

self.mysql\_config\_filepaths = []

self.mysql\_act\_config\_filepath = ''

self.config\_obj = self.create\_config\_obj(False)

self.mysql\_config\_obj = self.create\_mysql\_config\_obj()

self.mysql\_config\_header = 'mysqld'

def create\_mysql\_config\_obj(self):

self.find\_mysql\_config\_file()

config\_obj = self.create\_config\_obj(True)

print(self.mysql\_act\_config\_filepath)

print('hi')

return config\_obj

def create\_config\_obj(self, mysql):

print('hi')

config\_filepath = self.mysql\_act\_config\_filepath

print('hi')

dir\_path = os.path.dirname(os.path.realpath(self.curfile))

print('hi')

print(config\_filepath)

print('hi')

if mysql is False:

print('hi')

config\_filepath = dir\_path + self.data\_path + self.filename

self.config\_filepath = config\_filepath

print(config\_filepath)

print('hi')

print(config\_filepath)

exists = os.path.exists(config\_filepath)

config = None

if exists:

print("------------Config.ini exists at:", dir\_path + self.data\_path)

config = ConfigParser(allow\_no\_value=True)

print('hi')

config.read(config\_filepath, encoding='utf-8')

print('hi')

else:

raise Exception("------------Config.ini does not exists at:", dir\_path + self.data\_path)

return config

def refresh\_config\_obj(self):

config = ConfigParser()

config.read(self.config\_filepath, encoding='utf-8')

self.config\_obj = config

def return\_optionVal\_array(self, header, var):

if type(self.config\_obj[header][var]) == str:

clean\_array = self.Clean\_config\_arrays(header, var)

return clean\_array

else:

print(self.config\_obj[header][var], 'is type:', str(type(self.config\_obj[header][var]))+'.', 'Not <list>')

return None

def Clean\_config\_arrays(self, header, var):

try:

string = self.config\_obj[header][var]

string = string.replace(',', '')

string = string.replace('[', '')

string = string.replace(']', '')

array = string.split(' ')

print(array)

return array

except:

return None

def return\_value(self, header, var):

return self.config\_obj[header][var]

def write\_to\_config(self, header, keys, values):

available\_keys = list(self.config\_obj[header].keys())

try:

for i in range(len(keys)):

if keys[i] in available\_keys:

self.config\_obj[header][keys[i]] = str(values[i])

print(values[i])

else:

self.config\_obj[header].update({keys[i]:str(values[i])})

with open(self.config\_filepath, 'w') as file\_object:

self.config\_obj.write(file\_object)

except:

raise Exception("Number of keys and pairs do not math")

def find\_mysql\_config\_file(self):

#Create and array of all directories:

directories = os.path.dirname(self.mysql\_config\_default\_path).split("\\")[1:]

#Attempt1: Try to use the standard filepath for this (The configparser can hanlde .cnf and .ini files):

if os.path.exists(self.mysql\_config\_default\_path):

self.mysql\_config\_filepaths = [self.mysql\_config\_default\_path]

self.mysql\_act\_config\_filepath = self.mysql\_config\_default\_path

return True

#Attempt2: Go through the main MYSQL download location and look for it there:

paths1 = None

looked\_paths = []

if not file\_found:

for i in range(len(directories), 0, -1):

check\_looked\_at = True

bypass = False

if i >= len(directories):

check\_looked\_at = False

bypass = True

prev\_direct\_index = 'C:\\' + '\\'.join(directories[0:i + 1])

direct\_index = 'C:\\' + '\\'.join(directories[0:i])

difference\_index\_num = len(prev\_direct\_index)

paths2 = self.walk\_file\_tree(direct\_index, check\_looked\_at, difference\_index\_num, looked\_paths, bypass)

if paths2 and paths1:

paths1 += paths2

elif paths2 and not paths1:

paths1 = []

paths1 += paths2

looked\_paths.append(direct\_index)

if paths1:

self.mysql\_config\_filepaths = paths1

self.mysql\_act\_config\_filepath = paths1[0]

return paths1

return None

def walk\_file\_tree(self, directory, check\_looked\_at, difference\_index\_num, looked\_paths, bypass):

paths = None

tree = os.walk(directory)

for i, j, k in tree:

if check\_looked\_at and i[0:difference\_index\_num] not in looked\_paths or bypass:

if self.mysql\_config\_file in k and not paths:

paths = []

path = i + "\\" + self.mysql\_config\_file

paths.append(path)

elif self.mysql\_config\_file in k and paths and k not in paths:

path = i + "\\" + self.mysql\_config\_file

paths.append(path)

return paths

def set\_auto\_commit\_zero(self):

try:

self.mysql\_config\_obj[self.mysql\_config\_header]['autocommit'] = 0

assert os.path.isfile(self.mysql\_act\_config\_filepath)

with open(self.mysql\_act\_config\_filepath, 'w') as file\_object:

self.mysql\_config\_obj.write(file\_object)

except:

#raise Exception("python does not have permission to change the config file, or --: " + self.mysql\_act\_config\_filepath + " is not a file (it could be a directory, or a reference point)")

return False, 1

else:

return True, 0

#Test0: See if it works:

config\_test\_obj = config\_obj(data\_path, filename, curfile)

paths = config\_test\_obj.find\_mysql\_config\_file()

print(paths)

"""

#Tets1: Returns the users home directory:

print(os.environ)

print(os.environ['HOME'])

#Test2:

print(os.fspath('C:\ProgramData\MySQL\MySQL Server 8.0'))

print(os.fspath('my.ini'))

dict1 = {'PATH':'my.ini'}

print(os.get\_exec\_path(dict1))

"""

"""

#Test3: Add another file called 'my.ini. in the default directory:

config\_test\_obj = config\_obj(data\_path, filename, curfile)

paths = config\_test\_obj.find\_mysql\_config\_file()

print(paths)

#Test4: set autocommit to 0:

config\_test\_obj = config\_obj(data\_path, filename, curfile)

paths = config\_test\_obj.find\_mysql\_config\_file()

print(paths)

config\_test\_obj.set\_auto\_commit\_zero()

"""

###Python-MYSQL-User\_Interface###:

from find\_primes\_to\_array import find\_primes\_turn\_to\_array

from find\_primes\_to\_array import check\_prime

from find\_primes\_to\_array import find\_Eulers\_numbers

import Config\_handler

import random

import math

##Euler\_algor##:

data\_path = '/Data/'

filename = 'Config.ini'

curfile = str(\_\_file\_\_)

header = 'Euler'

var = 'primes'

#Math functions:

def cacl\_hcf(num1, num2):

hcf = 1

for i in range(1, min(num1, num2)+1):

if num1 % i == 0 and num2 % i == 0:

hcf = i

return hcf

#Euler Object:

class Euler\_algor\_obj:

def \_\_init\_\_(self, new\_or\_cur):

self.primes = find\_primes\_turn\_to\_array(data\_path, filename, curfile, header, var)

self.len\_primes = len(self.primes)

self.p = None

self.p\_index = None

self.q = None

self.n = None

self.thi = None

self.e = None

self.d = None

if new\_or\_cur:

self.p = self.calc\_p()[0]

self.p\_index = self.calc\_p()[1]

self.q = self.calc\_q()

self.n = self.calc\_n()

self.thi = self.calc\_thi()

self.e = self.calc\_e()

self.d = self.calc\_d()

else:

Eulers\_numbers\_import = find\_Eulers\_numbers(data\_path, filename, curfile, header)

self.p = Eulers\_numbers\_import['p']

self.q = Eulers\_numbers\_import['q']

self.n = Eulers\_numbers\_import['n']

self.thi = Eulers\_numbers\_import['thi']

self.e = Eulers\_numbers\_import['e']

self.d = Eulers\_numbers\_import['d']

self.publicKey = (self.e, self.n)

self.privateKey = (self.d, self.p, self.q)

self.org\_nums = []

self.cypher\_nums = []

self.config\_names = ['p', 'q', 'n', 'thi', 'e', 'd']

self.config\_values = [self.p, self.q, self.n, self.thi, self.e, self.d]

def set\_defaults(self):

self.p = self.defaults[0]

self.q = self.defaults[1]

self.n = self.defaults[2]

self.thi = self.defaults[3]

self.e = self.defaults[4]

self.d = self.defaults[5]

def calc\_p(self):

rand\_pos = random.randrange(self.len\_primes // 4, self.len\_primes // 2)

p = self.primes[rand\_pos]

return [p, rand\_pos]

def calc\_q(self):

q = self.primes[self.p\_index -1]

return q

def calc\_n(self):

n = self.p \* self.q

return n

def calc\_thi(self):

pn1 = self.p-1

qn1 = self.q-1

thi = pn1 \* qn1

return thi

def calc\_e(self):

e\_options\_array = []

for i in self.primes:

if i < self.thi and cacl\_hcf(self.thi, i) == 1 and cacl\_hcf(self.n, i) == 1:

e\_options\_array.append(i)

e = e\_options\_array[-1]

return e

def calc\_d(self):

d = self.e + 1

while (d \* self.e) % self.thi != 1 or not check\_prime(d):

d = d + 1

return d

def encrypt\_num(self, m):

#c = m^e mod n

c = (m \*\* self.e) % self.n

self.org\_nums.append(m)

self.cypher\_nums.append(c)

return c

def decrypt\_num(self, c):

#m = c^d mod n

m = (c \*\* self.d) % self.n

return m

def encrypt\_string(self, string):

letter\_ascii\_nums = [ord(i) for i in string]

encrypted\_letter\_ascii\_nums = [self.encrypt\_num(i) for i in letter\_ascii\_nums]

return encrypted\_letter\_ascii\_nums

def decrypt\_string(self, string\_array):

decrypted\_nums = [self.decrypt\_num(i) for i in string\_array]

decrypted\_nums\_to\_ascii\_letters = [chr(i) for i in decrypted\_nums]

substr = ''

for i in decrypted\_nums\_to\_ascii\_letters:

substr+=i

return substr

def write\_vars\_to\_config(self):

config\_write\_obj = Config\_handler.config\_obj(data\_path, filename, curfile)

config\_write\_obj.write\_to\_config(header, self.config\_names, self.config\_values)

"""

#test1.1:

Euler = Euler\_algor\_obj()

num = 30

print(num)

cypher = Euler.encrypt\_num(num)

print(cypher)

org\_num = Euler.decrypt\_num(cypher)

print(org\_num)

#test2:

Euler = Euler\_algor\_obj()

string = 'HiMyNameIsSlim\_Shady'

cypher = Euler.encrypt\_string(string)

org\_string = Euler.decrypt\_string(cypher)

"""

###Python-MYSQL-User\_Interface###:

from Config\_handler import config\_obj

import os.path

##Find Primes file - Turn to array

def check\_prime(primes\_array\_var):

prime\_checkers = [2,3,5,7,11]

#Quick test for a short list:

for i in prime\_checkers:

if primes\_array\_var not in prime\_checkers and primes\_array\_var % i == 0:

return False

return True

def cull\_non\_primes(primes\_array):

for i in range(len(primes\_array)):

is\_prime = check\_prime(primes\_array[i])

if not is\_prime:

primes\_array = primes\_array.pop(i)

return primes\_array

def find\_primes\_turn\_to\_array(data\_path, filename, curfile, header, var):

import\_config\_obj = config\_obj(data\_path, filename, curfile)

primes\_array = []

try:

primes\_array = list(map(int, import\_config\_obj.return\_optionVal\_array(header, var)))

except:

raise Exception("Not all of these are integers")

org\_primes\_array = primes\_array

primes\_array = cull\_non\_primes(primes\_array)

#Code to check if numbers have been removed: print(same\_list(org\_primes\_array, primes\_array))

return primes\_array

def same\_list(org, new):

if hash(str(org)) == hash(str(new)):

return True

return False

def find\_Eulers\_numbers(data\_path, filename, curfile, header):

Eulers\_numbers = {

'p':0,

'q':0,

'n':0,

'thi':0,

'e':0,

'd':0

}

import\_config\_obj = config\_obj(data\_path, filename, curfile)

items = list(import\_config\_obj.config\_obj[header].items())

for item in items:

if item[0] in Eulers\_numbers.keys():

Eulers\_numbers[item[0]] = int(item[1])

return Eulers\_numbers

###Python-MYSQL-User\_Interface###:

import pip

import importlib.util

import subprocess

import sys

import os

try:

from Config\_handler import config\_obj

except:

os.system('cmd /c "pip install Config\_handler"')

##Import\_python\_modules##:

installed\_packages = subprocess.check\_output([sys.executable, '-m', 'pip', 'freeze'])

flat\_installed\_packages = [r.decode().split('==')[0] for r in installed\_packages.split()]

print(flat\_installed\_packages)

data\_path = '/Data/'

filename = 'Config.ini'

curfile = str(\_\_file\_\_)

header = 'Modules'

var = 'Modules'

class imports\_handler:

def \_\_init\_\_(self):

self.modules\_array = self.Import\_python\_modules\_array()

self.file\_installed\_packages = []

self.uninstalled\_packages = []

def Import\_python\_modules\_array(self):

import\_config\_obj = config\_obj(data\_path, filename, curfile)

modules\_array = import\_config\_obj.return\_optionVal\_array(header, var)

if not modules\_array:

raise Exception("------------"+filename,"has no value",var,"at header",header)

return modules\_array

def Check\_modules\_installed(self):

for module in self.modules\_array:

if module in flat\_installed\_packages:

self.file\_installed\_packages.append(module)

else:

self.uninstalled\_packages.append(module)

if hash(str(self.file\_installed\_packages)) == hash(str(self.modules\_array)):

return True

return False

def add\_uninstalled\_module\_S(self, check):

if check:

self.Check\_modules\_installed()

for module in self.uninstalled\_packages:

self.add\_module\_S(module)

def install\_uninstalled\_module\_S(self, check):

if check:

self.Check\_modules\_installed()

for module in self.uninstalled\_packages:

self.install\_module(module)

def install\_module(self, module):

print(self.uninstalled\_packages)

try:

if hasattr(pip, 'main'):

pip.main(['install', module])

else:

pip.\_internal.main(['install', module])

except:

raise Exception("module:",module,". Does not exist")

def add\_module\_S(self, modules):

if type(modules) == str:

spec = importlib.util.find\_spec(modules)

module = importlib.util.module\_from\_spec(spec)

sys.modules[module] = module

spec.loader.exec\_module(module)

print(f"{module!r} has been imported")

elif type(modules) == list:

for module in modules:

self.add\_module\_S(module)

else:

raise Exception("type:",type(modules),"not accepted")

"""

#Test1:

i = imports\_handler()

print(i.modules\_array)

i.install\_uninstalled\_module\_S(True)

"""

###MYSQL-Python-User\_Interface###:

import Euler\_algor

import Config\_handler

import os

import csv

##Login system##

data\_path = '\\Data\\'

filename = 'Config.ini'

data\_path2 = '\\Data\\'

filename2 = 'user\_passwords.csv'

curfile = str(\_\_file\_\_)

def get\_user\_csv(filename3, data\_path3, curfile3):

path = os.path.realpath(os.path.dirname(curfile3)) + data\_path3 + filename3

user\_array = []

with open(path, newline = '') as csvfile:

csv\_data = csv.reader(csvfile, delimiter=',', quotechar='|')

user\_array = list(csv\_data)

return user\_array

def write\_user\_csv(filename3, data\_path3, curfile3, write\_data3):

path = os.path.realpath(os.path.dirname(curfile3)) + data\_path3 + filename3

fieldnames = ['username','password','host','database']

with open(path, 'a', newline = '') as csvfile:

csv\_writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

csv\_writer.writerow(write\_data3)

def get\_user\_csv\_dict(filename3, data\_path3, curfile3):

path = os.path.realpath(os.path.dirname(curfile3)) + data\_path3 + filename3

user\_dict = []

with open(path, newline = '') as csvfile:

csv\_data = csv.DictReader(csvfile)

for row in csv\_data:

user\_dict.append(row)

return user\_dict

def clean\_csv\_arrays(string):

string = string.replace(',', '')

string = string.replace('[', '')

string = string.replace(']', '')

string = string.split(' ')

array = list(map(int, string))

return array

def check\_Euler(config\_obj):

#Check Euler object has been created for the system:

header = 'log-in'

var = 'Euler\_created'

is\_Euler = bool(config\_obj.return\_value(header, var))

Euler\_obj = None

if not is\_Euler:

Euler\_obj = Euler\_algor.Euler\_algor\_obj(True)

Euler\_obj.write\_vars\_to\_config()

config\_obj.refresh\_config\_obj()

else:

Euler\_obj = Euler\_algor.Euler\_algor\_obj(False)

config\_obj.write\_to\_config('log-in', ['euler\_created'], ['1'])

return Euler\_obj

class session\_user():

def \_\_init\_\_(self, username, password, host, database):

self.username = ''

self.password = ''

self.host = ''

self.database = ''

self.connection\_obj = mysql\_connection(self.user, self.password, self.host, self.database)

def connect(self):

self.connection\_obj.setUp\_connection()

class login\_handler():

def \_\_init\_\_(self):

#Create the config object for the log-in system:

self.config\_obj = Config\_handler.config\_obj(data\_path, filename, curfile)

self.Euler\_obj = check\_Euler(self.config\_obj)

self.valid\_user = False

self.username = ''

self.password = ''

self.host = ''

self.database = ''

self.user\_obj = None

self.is\_user = False

def take\_username\_and\_password(self, username, password):

self.username = username

self.password = password

def check\_username\_password(self):

user\_found = False

password\_found = False

user\_array = get\_user\_csv\_dict(filename2, data\_path2, curfile)

for row in user\_array:

act\_user = self.Euler\_obj.decrypt\_string(clean\_csv\_arrays(row['username']))

act\_password = self.Euler\_obj.decrypt\_string(clean\_csv\_arrays(row['password']))

if act\_user == self.username and act\_password == self.password:

user\_found = password\_found = True

self.username = act\_user

self.password = act\_password

self.host = row['host']

self.database = row['database']

break

if user\_found and password\_found:

self.is\_user = True

return True

return False

def create\_seesion\_user(self):

if self.is\_user:

new\_user = session\_user(self.username, self.password, self.host, self.database)

self.user\_obj = new\_user

def return\_user(self):

return self.user\_obj

class create\_new\_user\_handler():

def \_\_init\_\_(self, username, password, host, database):

self.config\_obj = Config\_handler.config\_obj(data\_path, filename, curfile)

self.Euler\_obj = check\_Euler(self.config\_obj)

self.valid\_user = False

self.username = username

self.password = password

self.host = host

self.database = database

self.unacceptable\_characters = [':',';','.',',','[',']','{','}','/','?','|', '"\"', '"', "'", '=' ,'+', '#', '~', '@', '\*', '&', '^', '%', '$,' ,'£', '!']

self.csv\_data = {

'username': self.Euler\_obj.encrypt\_string(self.username),

'password': self.Euler\_obj.encrypt\_string(self.password),

'host': self.host,

'database': self.database

}

def check\_username(self):

acceptable\_user = True

already\_user = False

#Check acceptable:

for i in self.username:

if i in self.unacceptable\_characters:

acceptable\_user = False

#Check in use:

user\_array = get\_user\_csv\_dict(filename2, data\_path2, curfile)

for row in user\_array:

act\_user = self.Euler\_obj.decrypt\_string(clean\_csv\_arrays(row['username']))

if act\_user == self.username:

already\_user = True

if not acceptable\_user or already\_user:

return False

return True

def check\_connection(self):

connector = mysql\_connection(self.username, self.password, self.host, self.database)

is\_connected = connector.test\_connection()

return is\_connected

def write\_to\_users\_csv(self):

#Josh64,Joshua100,127.0.0.1,world

user\_dict = get\_user\_csv\_dict(filename2, data\_path2, curfile)

write\_user\_csv(filename2, data\_path2, curfile, self.csv\_data)

def create\_seesion\_user(self):

if self.is\_user:

new\_user = session\_user(self.username, self.password, self.host, self.database)

self.user\_obj = new\_user

"""

#Test1: Create User:

username = 'James24'

password = 'Joshua100x'

host = '127.0.0.2'

database = 'world'

user = create\_new\_user\_handler(username, password, host, database)

TF = user.check\_username()

print(TF)

#Test1:

login\_obj = login\_handler()

username = 'James24'

password = 'Joshua100T'

login\_obj.take\_username\_and\_password(username, password)

TF = login\_obj.check\_username\_password()

print(TF)

#Test2:

login\_obj = login\_handler()

username = 'James24'

password = 'Joshua100x'

login\_obj.take\_username\_and\_password(username, password)

TF = login\_obj.check\_username\_password()

print(TF)

"""