Cryptography Homework 2—Modular Arithmetic with Python

# Required Reading

Cryptology2 slides  
Alternative slides: Cryptology 0 Modular Arithmetic (much less math)  
Cracking Codes with Python by Sweigart  
 Chapter 13, pages 171 - 183 (or <https://inventwithpython.com/cracking/chapter13.html>) about modular arithmetic.  
 Chapter 5, pp. 54 - 67 (or <https://inventwithpython.com/cracking/chapter5.html>) about the Caesar cipher  
  
Optional Reading  
Wikipedia has good articles on modular arithmetic, greatest common divisor (GCD), Least Common Multiple (LCM), Euclid’s algorithm and Euclid’s extended algorithm.

# Optional Video

If you want to understand how Euclid’s extended algorithm works, this video by Christoff Paar is superb! <https://www.youtube.com/watch?v=fq6SXByItUI>.

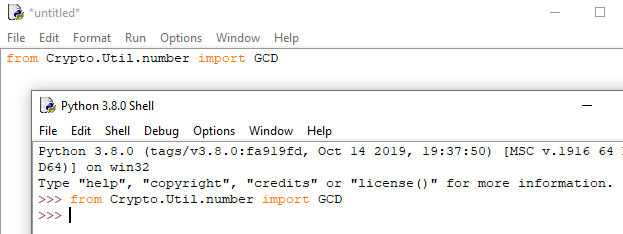
# Practicing Modular Math with PyCryptodome

We will use modules from the PyCryptodome package throughout the Cryptography lessons. In addition to modules for performing AES and RSA encryption, they have handy modules for modular arithmetic. Install PyCryptodome on the OS or VM you use by following the instructions in ‘PyCryptodome Installation.docx’. Note that Windows has recently made PyCryptodome more difficult by expanding their C++ compiler to over 5 GB, but it is still doable if you have the disk space; see ‘Problem Installing Pycryptodome on Windows 10.docx’.

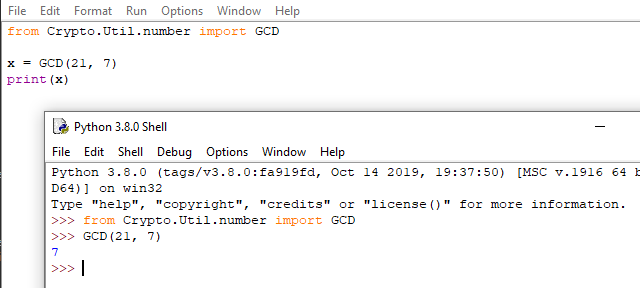
## Importing Python Modules

When Python starts, it does not include all the possible modules you might use. To save space, Python only includes the basics you need, and allows you to import additional modules as you need them. There are several ways to import modules, and the method you use determines the way you access the modules in your code. Unless I need many functions from a module, I prefer to import the modules by name, as shown below.

The functions we will use are included in PyCryptodome’s Crypto.Util package, more specifically in the Crypto.Util.number module. The function we will use today is the Greatest Common Divisor function, or GCD(x, y). It returns the greatest common divisor of the two inputs that you give it. Too make the GCD function available, include this line at the top of your code or run it in the interactive prompt. (Note: this will fail if PyCryptodome is not installed.)  
from Crypto.Util.number import GCD



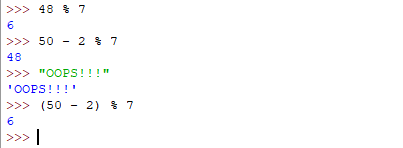
Here, I am showing a Python file in Idle, with a separate interactive prompt below. This shows both ways to use it.

Once that is done, you can use GCD() in your code or at the interactive prompt.  


## Python modular arithmetic operators

These operators are always included in Python, so they do not need to be imported.

### Modulus

The modulus operator in Python is “%”. The math statement 48 mod 7 is 48 % 7 in Python. If you have a slightly more complex statement, it is good to use parentheses to ensure that the modular operation is done last.  


### Integer Division

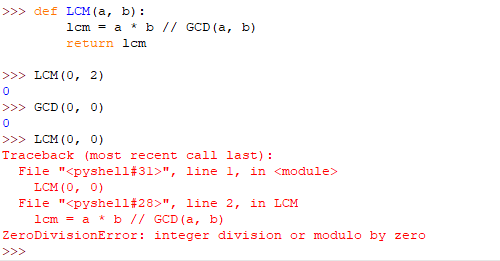
The division operator you are probably used to is “/”; it returns a real, or decimal, number.  

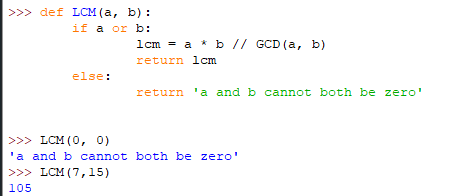

Usually in modular arithmetic we do not want the fractional part, so we use integer division. The integer division operator “//” truncates the fractional component or remainder and gives us an integer as a result. This is sometime called a floor function.  


### Creating your own function

We will often use the Least Common Multiple, but there is no function for that in PyCryptodome. You can choose to keep typing the formula for LCM into your code, or you can create your own function. Here is the formula:  

lcm(n_{1},\\; n_{2}) = {{n_{1} \\cdot n_{2} } \\over {gcd(n_{1},\\; n_{2})} }


Since we already have a GCD function, creating our own LCM function is easy. Here is a simple function, but you can see that it has a divide-by-zero problem.  


Here is an improved version.  


Note: The statement, if a or b:, is a shortcut. Any integer that is non-zero will evaluate as True, and zero will evaluate as False. So,  
if a or b:  
is the same as  
if (a != 0) or (b != 0):

## For Turn In

1. Compute (you can do this easily at an interactive Python prompt.)
   1. 34 mod 18
   2. (34 + 97) mod 12 Note: the mod operator has the same priority as multiplication. In a tie, Python executes operators from left to right.
   3. 14 \* 71 mod 15
   4. 152 / 71
   5. 152 // 71 (in Python, // is integer division)
   6. 152 – (152 // 71) \* 71 (this is the remainder after integer division)
   7. 152 mod 71 (you should see that the answer is the remainder when you divide 152//71 (integer division)
2. Compute gcd(36, 45) and gcd(44, 45) using the GCD function you imported above (from Crypto.Util.number import GCD). You should get 9 and 1 as answers, as a check to make sure the code is correct. Now compute gcd(452, 973) and gcd(452,1496). Which one of the pairs of numbers relatively prime, and what is the GCD of the pair that is not relatively prime?
3. For the following numbers, compute the GCD of the number and the modulus. If the number and the modulus are relatively prime, compute the multiplicative inverse. You can use either brute force in Python, or the inverse() function from PyCryptodome.  
   from Crypto.Util.number import inverse )
   1. 17 mod 26
   2. 16 mod 26
   3. Of the two numbers above, which could be used for the key of an affine cipher and which could not? Why?
   4. Compute the multiplicative inverse of 17 in modulus 27, 28, and 29. Note that the inverse is completely different when the modulus changes.
4. Least Common Multiple. The LCM(a, b) is the smallest number that can be divided by both   
   a and b. It is easily computed as a \* b // GCD(a, b)
   1. Compute the lcm of something simple, like 6 and 9. Check your answer by hand to verify that the equation is correct.
   2. Compute LCM(252, 196)
5. Use the file caesarCipher.py for this. The symbol set is expanded from the most common set, A – Z, and includes lower case, numbers, and some punctuation.   
   SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890 !?.'  
   The caesarCipher.py code has a problem, which you can see by encrypting a message with a key = 40. What happened? How can you fix it?
6. Use the file affineCipher.py (from *Cracking Codes with Python*) for this. It also used an expanded symbol set.  
   SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890 !?.'
   1. This symbol set is not a field, because not all elements have a multiplicative inverse. What caused that to happen? (Hint: What is the length of SYMBOLS?) What happens if you use a key with A = 2 and B = 13, or key = 135?
   2. Add or remove characters from SYMBOLS so that all elements of SYMBOLS have multiplicative inverses. Again len(SYMBOLS) is important; you are adjusting the length so that it is a \_\_\_\_\_ number.
      1. What SYMBOLS set did you choose, and what is its length?
      2. Encrypt a message. Hand in the encrypted message and the key you used.