Project Euler 100

Write-ups on the first 100 Project Euler problems.

I do not want to break policy, so I am only publicly posting the first 100.

- Euler 0028
- Euler 0029
- Euler 0030
- Euler 0032
- Euler 0033
- Euler 0034
- Euler 0035

Euler 0001

The Problem:

Listing all multiples of 3 and 5 under 1000.
Given: 3,5,6,9 are all multiples of 3 and 5 below 10.

Considerations:

We could iterate through every multiple of these numbers but there is going to be overlap when their multiples are divisible by both of them. Such examples would be 15 or 30. In these cases we don't want to add these numbers to the totals twice. This is what we need to watch out for in the code.

The Approach:

1. Keep iterating through all the multiples of 3 and 5
2. Add each number to a set, or a list with a not in conditional
3. Sum the set or list

The Code:

```
target_under = 1000
multiples = [3,5]
found = []

#go through each of our multiples
for multiple in multiples:
    #set the running total equal to the first multiple
    running_total = multiple
    #keep incrementing until we go over the target, then go to the next multiple
    while running_total < target_under:
        #add the running total to the list if it is not already in it.
        if running_total not in found:
            found += [running_total]
```

```
        running_total += multiple
```

Euler 0002

The Problem:

Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with and , the first terms will be: 1,2,3,5,8,13,21,34,55,89

By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.

What we know:

- We need to generate the Fibonacci sequence up to 4,000,000.
- We only need to keep every other number in the sequence.

Considerations:

Time-space considerations and decisions to be made:

- We could choose to store 0 numbers and recursively generate all numbers up to 4,000,000.
  - This spends both a lot of time although it doesn't use a large permanent space.
- We could generate all of the Fib. numbers up to 4,000,000 and store them in a list, good for referencing later, and will reduce redundancy.
- We could keep only the last 2 Fib. numbers and keep dropping the last one when we get a new one.

The Code:

```
fib_1 = 1
fib_2 = 2
limit = 4000000

#running total set to 2 because it is odd
running_total = 2

#keep going until fib_1 and fib_2 totaled > limit
```

```
while fib_1 + fib_2 < limit:
    fib = fib_1 + fib_2
    fib_1 = fib_2
    fib_2 = fib

    #add to the running total if it is even.
    if fib % 2 == 0:
        running_total += fib
```

Euler 0003

The Problem:

The prime factors of 13195 are 5, 7, 13, and 29.
What is the largest prime factor of the number 600951475143?

Considerations:

There are a lot of approaches, of which I will probably use later.

We are going to start with some basic sieving and then work out a more general algorithm:

Iterate from 2 to the upper limit:
- Is it divisible by 2:
    - add 2 to the prime factors list
    - set the loop to iterate up to upper_limit/2
        - Is it divisible by 2:
            - Keep doing above operation
- Is it divisble by 3:
    - add 3 to the prime factors list
    - set the loop to iterate up to upper_limit/3
- We don't care about 4.
- Is it divisible by 5:
    - add 5 to the prime factors list
    - set the loop to iterate up to upper_limit/5
- We don't care about 6.
- Is it divisible by 7:
    - add 5 to the prime factors list
    - set the loop to iterate up to upper_limit/7

etc.

Something to note: We don't care about any multiples of 2 and 3 after we check them, so a quick way to get around checking these multiples we can iterate through the loop by adding 6 and checking the before and after numbers.

The Code:

```
upper_limit = 6009515658416
#upper_limit = 20023110541 #, good to use to double check. Is 2 [2,2],3,167
current_upper_limit = upper_limit
prime_factors = []

#we are going to hardcode 2 and 3 for looping sake
if upper_limit % 2 == 0:
    while current_upper_limit % 2 == 0:
        current_upper_limit = current_upper_limit/2
        prime_factors += [2]
if upper_limit % 3 == 0:
    while current_upper_limit % 3 == 0:
        current_upper_limit = current_upper_limit/3
        prime_factors += [3]
current = 6

#go up to the upper limit of numbers, add 2 for edge case (I'm not sure that part is
correct)
while current < current_upper_limit + 2:
    #print(prime_factors, current_upper_limit)
    #print(prime_factors, current_upper_limit)
    #check the numbers before and after the increment of 6
    #add them to the prime factors as many times as necessary
    #keep dividing the new number to use.
    if current_upper_limit % (current - 1) == 0:
        while current_upper_limit % (current - 1) == 0:
            current_upper_limit = current_upper_limit/(current-1)
            prime_factors += [current-1]
    if current_upper_limit % (current + 1) == 0:
        while current_upper_limit % (current + 1) == 0:
            current_upper_limit = current_upper_limit/(current+1)
            prime_factors += [current+1]
    current += 6

print(prime_factors)
```

The Twist:

While I was testing this it took insanely long because I was calculating for 600951475143 instead of 600851475143.

The largest prime factor of 600**8**51475143 is 6857. This is not very high, and easy enough to brute force.
The largest prime factor of 600**9**51475143 is 1552846189. This is a lot of iterations to get too and the code would timeout.

After reading the solution provided by Project Euler I understood what I was missing to get the solution to this typo'd part.

Every number can have at most one prime factor that is greater than its square root.
I have modified the code below to reflect this change and break out of the while loop if the current_upper_limit is greater than the square root of the original number.

This means that we only have to iterate through 775210 (square root of 600951475143) before we (can rightfully) call it quits and accept that:
- This is prime
- This is the last prime factor

Updated Code:

```
import math

upper_limit = 600951565841652
#upper_limit = 20023110541 #, good to use to double check. Is [2,2,3,167]
current_upper_limit = upper_limit
sqrt_limit = math.sqrt(upper_limit)
prime_factors = []

#we are going to hardcode 2 and 3 for looping sake
if upper_limit % 2 == 0:
    while current_upper_limit % 2 == 0:
        current_upper_limit = current_upper_limit/2
        prime_factors += [2]
if upper_limit % 3 == 0:
    while current_upper_limit % 3 == 0:
        current_upper_limit = current_upper_limit/3
        prime_factors += [3]
current = 6
```

```
#go up to the square root
while current < sqrt_limit:
    #print(prime_factors, current_upper_limit)
    #check the numbers before and after the increment of 6
    #add them to the prime factors as many times as necessary
    #keep dividing the new number to use.
    if current_upper_limit % (current - 1) == 0:
        while current_upper_limit % (current - 1) == 0:
            current_upper_limit = current_upper_limit/(current-1)
            prime_factors += [current-1]
    if current_upper_limit % (current + 1) == 0:
        while current_upper_limit % (current + 1) == 0:
            current_upper_limit = current_upper_limit/(current+1)
            prime_factors += [current+1]
    current += 6

#if we looped beyond the square root limit and the number
#we are left with isn't 1, then it is the final prime factor.
if current_upper_limit != 1:
    prime_factors += [current_upper_limit]

print(prime_factors)
```

Euler 0004

The Problem:

9009 is a palindromic number since it can be read forwards and backwards the same. It is the product of 91*99 and is the largest palindromic number generated by the product of 2 2-digit numbers.
What is the largest palindromic number that is the product of 2 3-digit numbers?

The Approach:

The incredibly naive approach would be to start with 999*999 and iterate through all iterations going down through all the numbers 999->1 and 999->1. We are probably going to find the number pretty near the top, but still running the entire brute force is *only* 998001 iterations... which isn't *too* bad.

As for the checker, we can convert the number into a string and check if it the same forwards and backwards.

The Code:

```
largest_palindrome = 0


def palindrome(number):
    return str(number) == str(number)[::-1]


for i in range(1000,1,-1):
    for j in range(1000,1,-1):
        if palindrome(i*j) and i*j > largest_palindrome:
            largest_palindrome = i*j


print(largest_palindrome)
```

Euler 0005

The Problem:

2520 is the smallest number evenly divisible by the numbers 1 through 10.
What is the smallest number evenly divisible by the numbers 1 through 20?

Considerations:

With the context provided we now know:
- That our function should produce 2520 with an upper limit of 10.
- The number we are looking for is less than the factorial of the upper limit.

The Approach:

One way we can approach this is to get the union of all the prime factors of each number.
There will be redundancy in this approach, but it should be performant enough.

Just listing out the first 6 numbers we should see:
- 2 [2]
- 3 [3]
- 4 [2,2]
- 5 [5]
- 6 [2,3]

Which should mean that the function of 6 would produce 2*2*3*5 or 60.

I'll modify the prime factorizing method from Euler 3 to just produce prime factors.

The Code:

```
import math


smallest_multiple = []
upper_limit = 20


def prime_factors(upper_limit):
```

```
    #upper_limit = 20023110541 #, good to use to double check. Is [2,2,3,167]
    current_upper_limit = upper_limit
    sqrt_limit = math.sqrt(upper_limit)
    prime_factors = []

    #we are going to hardcode 2 and 3 for looping sake
    if upper_limit % 2 == 0:
        while current_upper_limit % 2 == 0:
            current_upper_limit = current_upper_limit/2
            prime_factors += [2]
    if upper_limit % 3 == 0:
        while current_upper_limit % 3 == 0:
            current_upper_limit = current_upper_limit/3
            prime_factors += [3]
    current = 6

    while current < sqrt_limit:
        #print(prime_factors, current_upper_limit)
        if current_upper_limit % (current - 1) == 0:
            while current_upper_limit % (current - 1) == 0:
                current_upper_limit = current_upper_limit/(current-1)
                prime_factors += [current-1]
        if current_upper_limit % (current + 1) == 0:
            while current_upper_limit % (current + 1) == 0:
                current_upper_limit = current_upper_limit/(current+1)
                prime_factors += [current+1]
        current += 6

    if current_upper_limit != 1:
        prime_factors += [current_upper_limit]

    return prime_factors

#start from 2 and go to the number we want to find
for i in range(2,upper_limit + 1,1):
    current_pf = prime_factors(i)
    new_smallest_multiple = []
    #look at all of the numbers between our current set and what we just factorized
    for number in set(smallest_multiple).union(set(current_pf)):
```

```
        #add that many prime factors to the current smallest multiple
        for j in range(max(smallest_multiple.count(number),current_pf.count(number))):
            new_smallest_multiple += [number]
    smallest_multiple = new_smallest_multiple

print(smallest_multiple)
print(math.prod(smallest_multiple))
```

Euler 0006

The Problem:

The sum of squares n is defined by the sum of natural numbers ascending, such that, sum_of_squares(10) is 385. 1^2 + 2^2 + ... 10 ^2.
The square of sum n is defined by the square of the sum of natural numbers ascending, such that, square_of_sum(10) is 3025. (1+2+...+10)^2 = 3025
Find the difference of these 2 values where n is 100.

Considerations:

Funny enough, this doesn't even need iterative computation, there are 2 different formula we can work with:
- Sum of square: n(n+1)(2n+1)/6
- Sum of numbers (and then squared): (n(n+1)/2)^2

The Solution (No-Code!):

Which means that the final formula is (n(n+1)/2)^2 - n(n+1)(2n+1)/6
Which by hand:
- (100(101)/2)^2 - 100(101)(201)/6
- 25502500 - 338350
- 25164150

Euler 0007

The Problem:

Prime numbers are numbers that are only divisible by themselves and 1. Let's generate primes!
What is the 1001st prime?

The Approach:

We can re-use some of the prime factorization code, and just check the stacked prime factors that we have accumulated.

We start with the primes 2 and 3 and just move up numbers up through numbers by 6 and check each neighbor. This mathematically works out because of basic rules of sieving.

It is definitely not an optimal algorithm, but should be good enough for 1000 primes. So I'm happy with that.

The Code:

```
prime_count = 10001
prime_factors = [2,3] #we can just start with 2 and 3, because we already know them

#simple prime checker
def add_if_prime(number, primes):
    is_prime = True
    #check through all the primes we have gathered so far.
    for prime in primes:
        if number % prime == 0:
            is_prime = False
            break

    if is_prime:
        primes += [number]

    return primes
```

```
#actual loop to iterate through
current = 6
while len(prime_factors) < prime_count:
    add_if_prime(current-1, prime_factors) #check the number before
    add_if_prime(current+1, prime_factors) #check the number after
    current += 6 #loop through incrementing by 6

print(prime_factors[prime_count-1])
```

Euler 0008

The Problem:

In this giant 1000 digit number the greatest product of 4 adjacent numbers is 9*9*8*9 = 5832
We need to find the greatest 10 adjacent numbers

7316717653133062491922511967442657474235534919493
4
9698352031277450632623957831801698480186947885184
3
8586156078911294949545950173795833195285320880551
1
1254069874715852386305071569329096329522744304355
7
6689664895044524452316173185640309871112172238311
3
6222989342338030813533627661428280644448664523874
9
3035890729629049156044077239071381051585930796086
6
7017242712188399879790879227492190169972088809377
6
6572733300105336788122023542180975125454059475224
3
5258490771167055601360483958644670632441572215539
7
5369781797784617406495514929086256932197846862248
2
8397224137565705605749026140797296865241453510047
4
8216637048440319989000889524345065854122758866688
1
1642717147992444292823086346567481391912316282458
6
1786645835912456652947654568284891288314260769004
2
2421902267105562632111110937054421750694165896040
8
0719840385096245544436298123098787992724428490918
8
8458015616609791913387549920052406368991256071760
6
0588611646710940507754100225698315520005593572972
5
7163626956188267042825248360082325753042075296345
0

The Approach:

We can just iterate through... 1 thousand (minus 10) calculations is ultimately not much for the computer.

To optimize a little better though, we can make assumptions with the window. When we move the window across, if the number going out is larger than the number going in then we don't need to run a calculation... it is going to be smaller than our current max.
Otherwise we will run the calculation.

The Code:

```
```

```
import math

large_number =
"73167176531330624919225119674426574742355349194934969835203127745063262395783180169848018
69478851843858615607891129494954595017379583319528532088055111254069874715852386305071569 3
29096329522744304355766896648950445244523161731856403098711121722383113622298934233803081 3
53362766142828064444866452387493035890729629049156044077239071381051585930796086670172427 1
21883998797908792274921901699720888093776657273330010533678812202354218097512545405947522 4
35258490771167055601360483958644670632441572215539753697817977846174064955149290862569321 9
78468622482839722413756570560574902614079729686524145351004748216637048440319989000889524 3
45065854122758866688116427171479924442928230863465674813919123162824586178664583591245665 2
94765456828489128831426076900422421902267105562632111110937054421750694165896040807198403 8
50962455444362981230987879927244284909188845801561660979191338754992005240636899125607176 0
60588611646710940507754100225698315520005593572972571636269561882670428252483600823257530 4
20752963450"
window_size = 13
best_index = 0
best_index_value = math.prod([int(x) for x in large_number[0:window_size]])

for i in range(1,len(large_number) - window_size):
    if int(large_number[i-1]) < int(large_number[i+window_size-1]):
        current_value = math.prod([int(x) for x in large_number[i:i+window_size]])
        if current_value > best_index_value:
            best_index_value = current_value
            best_index = i
            print("newly found at ", best_index, best_index_value)

print(large_number[best_index:best_index+window_size])
```

Euler 0009

The Problem:

A pythagorean triple is such that integers a, b, c can be a^2 + b^2 = c^2.

There exists a pythagorean triple that sum total of a + b + c = 1000. Find it.

The Approach:

As a relatively naive approach:
- We can iterate through a and b and take the sum of squares and then the square root of the result to produce c
- We add a + b + c:
    - If the sum is the goal then we break
    - If the sum is greater than the goal then we break and continue with an increment
    - If the sum is less than the goal we continue

The Code:

```
import math

number_to_find = 1000
a = 0
b = 0
c = 0

#we will never reach the end, but it to prevent it from going infinite
for i in range(1,number_to_find):
    for j in range(i,number_to_find): #we can start from i because we technically already
calculated the reverse of the pair
        c =  math.sqrt(i**2 + j**2)
        if int(c) == c: #if it is actually an integer, we will continue
            if i + j + c > number_to_find:
```

```
                break
            elif i + j + int(c) == number_to_find:
                a = i
                b = j
                break #don't forget this break, it's crucial, lol

    if a != 0: #we can break out if we found the triple
        break

print(a,b,c)
print(a+b+c)
print(a*b*c)
print(a**2+b**2, c**2)
```

Euler 0010

The Problem:

We are finding primes again, this time all the primes up to 2000000!

We are going to modify the code that was created in Euler Problem 7 to generate up to a number instead of up to a prime count.
Then it is a simple matter of summing up all of the primes.

The issue... the last prime generator took 2.4 seconds to generate up to 100,000. So.. It's probably not going to make the Euler 1 minute guarantee if we are going to 20 times the number.

Considerations:

Yeah, I tried the more brutish generator and it didn't work... I'm going to try a different and possibly even stranger approach.

I am going to represent the threshold that we want to get to as an array of numbers. This does mean I am trading computation for space... but space is cheap.

The Approach:

We will start with an array of 2000000 and then cull out 0 and 1.
Then we start at 2, ignore it, and mark everything that is a multiple of 2.
Then we go to 3, ignore it, and mark everything that is a multiple of 3.
We will keep going until there are no more numbers to check.

Basically, it's the sieve of Eratosthenes.

The Code:

```
prime_threshold = 2000000
number_list = list(range(2, prime_threshold+1)) #initiating the sieve. Lot's of space here
on startup being used
prime_list = []
```

```
current = 0
total_iteration = 0
while current < len(number_list): #while we haven't iterated through the whole sieve yet
    if number_list[current] != -1: #find the next number that isn't marked
        prime = number_list[current]
        prime_list += [prime] #add it to our prime list, because it is prime.

        increment = current + prime
        while increment < len(number_list): #and go through the rest of the sieve marking
numbers with it
            number_list[increment] = -1
            increment += prime
            total_iteration += 1
    current += 1

print(len(prime_list)) #look at the length of the prime list, this is the number of primes
under prime_threshold
print(total_iteration) #the amount of iterations that we made, much better than
recalculating primes for each number
print(sum(prime_list)) #the answer
```

Euler 0011

The Problem:

Find the greatest product of 4 adjacent numbers in the grid (which can be found on the PE website).
This can be up, down, left, right, or diagonal.

Considerations:

- We don't need to check left ever. In the last 3 columns we don't check right, either.
- We don't need to check up ever. In the last 3 rows we don't check down.
- We don't need to check up diagonals. We will check both left and right diagonals, and ignore the relevant diagonals when we are in the first or last 3 columns. We don't need to check diagonals in the last 3 rows.

The Approach:

Starting from 0,0 being the topmost leftmost corner of the grid, we can keep iterating through and checking every case, remembering the considerations above.

The Code:

```
import numpy
from enum import Enum

class Direction(Enum): #not necessary to the solution, but helps me as a human visualize
where the solution is at
    DOWN = "DOWN"
    RIGHT = "RIGHT"
    LEFT_DIAG = "LEFT_DIAG"
    RIGHT_DIAG = "RIGHT_DIAG"

best_index = [0,0]
best_dir = Direction.DOWN
```

```
best_product = 0
length = 4
grid = numpy.genfromtxt("numbers", delimiter=" ") #using numpy to read in the file to a
grid for us

for i in range(len(grid)): #go through every row in the grid
    for j in range(len(grid[i])): #go through every column in the grid
        #print(i,j)
        #check right
        if j < len(grid)-(length-1):
            current_prod = 1
            for k in range(length):
                current_prod *= grid[i,j+k]
            if current_prod >= best_product:
                best_index = [i,j]
                best_dir = Direction.RIGHT
                best_product = current_prod

        #check down
        if i < len(grid)-(length-1):
            current_prod = 1
            for k in range(length):
                current_prod *= grid[i+k,j]
            if current_prod >= best_product:
                best_index = [i,j]
                best_dir = Direction.DOWN
                best_product = current_prod

        #check down right
        if i < len(grid)-(length-1) and j < len(grid)-(length-1):
            current_prod = 1
            for k in range(length):
                current_prod *= grid[i+k,j+k]
            if current_prod >= best_product:
                best_index = [i,j]
                best_dir = Direction.RIGHT_DIAG
                best_product = current_prod

        #check down left
```

```
        if i < len(grid)-(length-1) and j > (length-1):
            current_prod = 1
            for k in range(length):
                current_prod *= grid[i+k,j-k]
            if current_prod >= best_product:
                best_index = [i,j]
                best_dir = Direction.LEFT_DIAG
                best_product = current_prod

print(best_index, best_dir, best_product) #prints out the best position, the direction it points, and the product
```

Euler 0012

The Problem:

What is the value of the first triangle number to have over five hundred divisors?

Considerations:

We don't actually need to compute all of the divisors of a number, since the number of divisors can be calculated by knowing the number of prime factors.

12, for example, has prime factors 2(2 of these) and 3. It has a total of 6 divisors, which can be calculated by taking the quantity of each prime factor and adding 1, 2 for 2 becomes 3 and 1 for 3 becomes 2... Ok, maybe that's an obtuse way of explaining it... If you want a good explanation of it, you can find it here: https://mathschallenge.net/library/number/number_of_divisors

The Approach:

We are going to use the same prime factors finder that we have before, but instead of we are going to multiple by a running divisors count.

The Code:

```
import math

#This is a modification of the prime factor finder
def calculate_divisors(number):
    upper_limit = number
    current_upper_limit = upper_limit
    sqrt_limit = math.sqrt(upper_limit)
    current_divisors = 1

    #we are going to hardcode 2 and 3 for looping sake
    if upper_limit % 2 == 0:
        current_div = 1
```

```
        while current_upper_limit % 2 == 0:
            current_upper_limit = current_upper_limit/2
            current_div += 1
        current_divisors *= current_div
    if upper_limit % 3 == 0:
        current_div = 1
        while current_upper_limit % 3 == 0:
            current_upper_limit = current_upper_limit/3
            current_div += 1
        current_divisors *= current_div
    current = 6

    while current < sqrt_limit:
        #print(prime_factors, current_upper_limit)
        if current_upper_limit % (current - 1) == 0:
            current_div = 1
            while current_upper_limit % (current - 1) == 0:
                current_upper_limit = current_upper_limit/(current-1)
                current_div += 1
            current_divisors *= current_div
        if current_upper_limit % (current + 1) == 0:
            current_div = 1
            while current_upper_limit % (current + 1) == 0:
                current_upper_limit = current_upper_limit/(current+1)
                current_div += 1
            current_divisors *= current_div
        current += 6

    return current_divisors

bailout = 100000 #keep checking up to a bailout number

triangle = 1
for i in range(2,bailout):
    triangle += i
    if calculate_divisors(triangle) >= 500:
        break

print(i, triangle)
```

Euler 0013

The Problem:

There is 100 50-digit numbers that needed to be summed together. The attached file has been added to do this.

Considerations:

In python this problem is irrelevant because we can fit giant numbers into the integer space and python will work with it.

The Approach:

We read the file, sum the numbers, convert to string, and then convert back into number after slicing the first 10 digits.

The Code:

```
numbers_file = open("numbers.txt", "r")


total = 0
for number in numbers_file:
    total += int(number)


print(str(total)[:10])
```

Euler 0014

The Problem:

We are trying to find the longest Collatz sequence under 1000000.

Considerations:

The collatz sequence follows 2 simple rules:
- if n is even, then n/2
- if n is odd, then n*3 + 1

5 -> 16 -> 8 -> 4 -> 2 -> 1
6 -> 5 -> 4 -> 3 -> 2 -> 1

The Approach:

The approach I'll take to this is to set up a dictionary of lengths at key n. If the key doesn't have a value then it generates a collatz sequence until it returns 1 or a key that has a length value

The Code:

```
highest_starting =  1000000
lengths_dict = {1:1}
max_length = 1
length_key = 1

#the recursive function to generate collatz.
#Base case is satisfied by the lengths_dict above already having 1 initialized.
def update_lengths(number, lengths):
    #print(number)
    if lengths.get(number) == None:
        if number % 2 == 0:
```

```
            one_down = number/2
        else:
            one_down = number*3 + 1
        lengths = update_lengths(one_down, lengths)
        lengths[number] = lengths[one_down] + 1

    return lengths

for i in range(1,highest_starting + 1):
    if lengths_dict.get(i) == None:
        lengths_dict = update_lengths(i, lengths_dict)

    if lengths_dict.get(i) > max_length:
        length_key = i
        max_length = lengths_dict.get(i)

print(length_key, max_length)
```

Euler 0015

The Problem:

How many paths can you take if you just walk through a grid by only moving down and right? We are looking for 20x20. 2x2 has a given value of 6.

Considerations:

As it turns out, this is just the max value of all the odd layers of pascal's triangle.
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1

The Approach:

So if we generate 41 rows of pascal then we can take the max value and that is the answer.

The Code:

```
pascal_rows = 41

def generate_pascals_last_row(rows):
    last_row = [1]
    #we have already generated row 1
    for i in range(rows-1):
        #take the first number (hint, it's 1)
        current_row = [1]

        #add up all the numbers and put them in between
        for j in range(0, len(last_row) - 1):
```

```
            current_row += [last_row[j] + last_row[j+1]]

        #take the last number (hint, it's 1)
        current_row += [1]
        last_row = current_row

    return last_row

print(max(generate_pascals_last_row(pascal_rows)))
```

Euler 0016

The Problem:

What is the sum of the digits of 2^1000?

Considerations:

With python we can trivially calculate 2^1000 and then we can convert it to a string and sum its digits.

The Approach:

We calculate 2^1000 and then we can convert it to a string and sum its digits.

The Code:

```
power = 1000
number = 2**power

total_sum = 0
for digit in str(number):
    total_sum += int(digit)

print(total_sum)
```

Euler 0017

The Problem:

If you were to write out a number as a word, you could then count the letters and generate a new number.
For example: 1 becomes one which becomes 3.
We use british convention which adds an 'and' between the hundreds and tens place. For example one hundred and forty-four.
How many letters are used in total to write out all of the numbers from 1 to 1000? We can exclude hyphens.

Considerations and Approach:

There are a lot of uniques we need to keep track of, but then we can convert numbers off of a set of rules.

1 - one - 3
2 - two - 3
3 - three - 5
4 - four - 4
5 - five - 4
6 - six - 3
7 - seven - 5
8 - eight - 5
9 - nine - 4
10 - ten - 3
11 - eleven - 6
12 - twelve - 6
13 - thirteen - 8
14 - fourteen - 8
15 - fifteen - 7
16 - sixteen - 7
17 - seventeen - 9
18 - eighteen - 8
19 - nineteen - 8
20 - twenty - 6
30 - thirty - 6
40 - forty - 5
50 - fifty - 5
60 - sixty - 5
70 - seventy - 7

80 - eighty - 6
90 - ninety - 6
and - 3
thousand - 8
hundred - 7

1 through twenty have distinct numbers so we will need to keep that into account.

The Code:

```
number_dict =
{1:3,2:3,3:5,4:4,5:4,6:3,7:5,8:5,9:4,10:3,11:6,12:6,13:8,14:8,15:7,16:7,17:9,18:8,19:8,20:
6,30:6,40:5,50:5,60:5,70:7,80:6,90:6}
number_end = 1000

total_sum = 0
for i in range(1,number_end + 1):
    construction = ""
    hundred = False
    if int(i / 1000) > 0:
        total_sum += number_dict[int(i / 1000)] #thousands place number
        #print(i,int(i / 1000))
        total_sum += 8 #the word thousand
    if int((i % 1000) / 100) > 0:
        #print(i,int((i % 1000) / 100))
        total_sum += number_dict[int((i % 1000) / 100)] #hundreds place number
        total_sum += 7 #the word hundred
        hundred = True
        construction += str(int((i % 1000) / 100)) + "hundred"
    if int((i % 100)) > 0:
        tens = int((i % 100)/10)*10
        ones = int((i % 100))
        ones_true = int(i%10)
        if hundred:
            construction += " and "
```

```
            total_sum += 3 #the word and
        if tens >= 20: #numbers greater than 20
            construction += str(tens)
            total_sum += number_dict[tens]
        if ones > 9 and tens < 20: #10 through 19
            construction += str(ones)
            total_sum += number_dict[ones]
        if ones_true > 0 and (ones < 10 or ones > 19): #ten through 19 are weird
            construction += str(ones_true)
            total_sum += number_dict[ones_true]
    #print(construction)

print(total_sum)
```

Euler 0018

The Problem:

Moving through a triangle of number (every number chosen yields 2 more numbers to choose), what is the highest sum path in the triangle?

Considerations and Approach:

We will populate a tree data structure with the data from the triangle, and then work from the bottom up to find the best path.

Starting at the bottom of the tree take the value of the nodes themself, then when moving up the tree, take the best value from the left and right children and add it to the sum.

This results in the top node having the best sum.

The Code:

```
triangle_file = open("triangle", "r")

class Node:
    def __init__(self, value):
        self.l_node = None
        self.r_node = None

        self.value = value
        self.l_sum = -1
        self.r_sum = -1
        self.best = -1

#not necessarily the prettiest way to construct a tree... but oh well.
#there is a lot by reference here.
top_triangle = int(triangle_file.readline())
triangle_tree = Node(top_triangle)
```

```
past_line = [triangle_tree]
for line in triangle_file:
    current_line = [Node(int(x)) for x in line.split()]
    for i in range(len(past_line)):
        past_line[i].l_node = current_line[i]
        past_line[i].r_node = current_line[i+1]
    past_line = current_line

#make a function to populate the sums
def pop_sums(node : Node):
    if node.l_node is None:
        node.l_sum = node.value
        node.r_sum = node.value
        node.best = node.value
    else:
        #make sure both below are populated
        if node.l_node.best == -1:
            pop_sums(node.l_node)
        if node.r_node.best == -1:
            pop_sums(node.r_node)

        node.l_sum = node.l_node.best + node.value
        node.r_sum = node.r_node.best + node.value
        node.best = node.l_sum if node.l_sum > node.r_sum else node.r_sum

pop_sums(triangle_tree)

print(triangle_tree.best)
```

Euler 0019

The Problem:

How many Sundays fell on the first of the month between Jan 1 1901 and Dec 31 2000?

Considerations:

There is for sure a number theory mathematical way of handling this... but...

Approach:

What if we just imported a python calendar module and counted how many Sundays were between then and now?

The Code:

```
import calendar

year = 1901
end_year = 2000

sunday_first_days = 0
for i in range(year, end_year + 1):
    this_year = calendar.Calendar().yeardayscalendar(i,12)
    for year in this_year:
        for month in year:
            for week in month:
                if week[6] == 1:
                    sunday_first_days += 1

print(sunday_first_days)
```

Euler 0020

The Problem:

What is the sum of the digits of 100! where n! means 1x2x3x4x...xn?

Considerations and Approach:

For Python this is trivial to produce 100! and then take the sum of the digits by converting to a string and back.

The Code:

```
import math

factorial = 100

number = sum([int(x) for x in str(math.factorial(factorial))])

print(number)
```

Euler 0021

The Problem:

Let d(n) be defined as the sum of proper divisors of (numbers less than n which divide evenly into n).
If d(a) = b and d(b) = a, where a =/= b, then a and b are an amicable pair and each of a and b are called amicable numbers.

For example, the proper divisors of 220 are 1,2,4,5,10,11,20,22,44,55 and 110; therefore d(220) = 284. The proper divisors of 284 are 1,2,4,71 and 142; so d(285) = 220.

Evaluate the sum of all the amicable numbers under 10000.

Considerations and Approach:

We can generate divisors up to 10,000 for each number using a modified Euler 3.
We can store a dictionary of sums (this is probably unoptimal) as we go along.
When we store a sum, we can check if the sum is already in the dictionary. If it is and the dictionary value for that sum is the original number, we can add both the number and the sum into the dictionary.

The Code:

```
import math

#generating the primes beforehand is incredibly helpful, since we really don't need to re-
brute force these
def generate_primes(upper):
    prime_threshold = upper
    number_list = list(range(2, prime_threshold+1))
    prime_list = []

    current = 0
    total_iteration = 0
    while current < len(number_list):
```

```
        if number_list[current] != -1:
            prime = number_list[current]
            prime_list += [prime]

            increment = current + prime
            while increment < len(number_list):
                number_list[increment] = -1
                increment += prime
                total_iteration += 1

        #print(number_list)
        #print(prime_list)
        current += 1

    return prime_list

def divisor_finder(number, primes):
    upper_limit = number
    #upper_limit = 20023110541 #, good to use to double check. Is [2,2,3,167]
    current_upper_limit = upper_limit
    sqrt_limit = math.sqrt(upper_limit)
    prime_factors = []

    #we are going to hardcode 2 and 3 for looping sake
    i = 0
    while primes[i] < sqrt_limit and i < len(primes):
        while current_upper_limit % primes[i] == 0:
            current_upper_limit = current_upper_limit/primes[i]
            prime_factors += [primes[i]]
        i += 1

    if current_upper_limit != 1:
        prime_factors += [current_upper_limit]

    #print(prime_factors)

    prime_counts = [[prime_factors[0], prime_factors.count(prime_factors[0])]]
    #print(prime_counts,prime_counts[-1][0],prime_factors[1])
    for i in range(1,len(prime_factors)):
```

```
        #print(prime_counts[-1][0],prime_factors[i])
        if  prime_counts[-1][0] != prime_factors[i]:
            prime_counts += [[prime_factors[i], prime_factors.count(prime_factors[i])]]

    #print(prime_counts)

    divisors = [1]
    counters = [x[1] for x in prime_counts]
    #print(counters)
    while sum(counters) > 0:
        new_divisor = 1
        for i in range(len(counters)):
            #print(prime_counts[i][0],counters[i], new_divisor)
            new_divisor *= prime_counts[i][0]**counters[i]
        divisors += [new_divisor]

        counters[0] -= 1
        for i in range(len(counters)):
            if counters[i] == -1:
                counters[i] = prime_counts[i][1]
                if i != len(counters) - 1:
                    counters[i+1] -= 1
            else:
                break

    divisors.sort()
    return divisors[:-1]



upper = 10000
prime_list = generate_primes(upper)
#print(prime_list)
all_amicables = []
number_sums = {}
for y in range(2,upper):
    number_sums[y] = sum(divisor_finder(y,prime_list))
    if number_sums[y] in number_sums:
        if number_sums[y] != y:
            if number_sums[number_sums[y]] == y:
```

```
                all_amicables += [number_sums[y], y]



# print(divisor_finder(220), divisor_finder(284))
# print(sum(divisor_finder(220)), sum(divisor_finder(284)))
print(number_sums[220],number_sums[284])
print(all_amicables)
print(sum(all_amicables))
```

Euler 0022

The Problem:

Get the total sum of a file of names if each letter in the name is mapped from 1-26 and then multiplied by the position in an alphabetical sort.

Considerations and Approach:

We read in the file into an array of names. After that it just needs to be sorted and then propagated through to add the letter sum multiplied by the alphabetical position to the total sum.

The Code:

```
names = open("names.txt", "r")

name_list = sorted([x[1:-1] for x in names.readline().split(",")])

names_score = 0
for i in range(len(name_list)):
    current_name = name_list[i]
    current_score = 0
    for letter in current_name:
        #print(letter, ord(letter) - 64)
        current_score += ord(letter) - 64

    names_score += current_score*(i+1)
    print(name_list[i], (i+1)*current_score, names_score)

print(len(name_list))
```

Euler 0023

The Problem:

A perfect number is a number for which the sum of its proper divisors is exactly equal to the number.
A number is called deficient if the sum of its proper divisors is less than and it is called abundant if this sum exceeds.

As 12 is the smallest abundant number, the smallest number that can be written as the sum of two abundant numbers is 24. By mathematical analysis, it can be shown that all integers greater than 28123 can be written as the sum of two abundant numbers. However, this upper limit cannot be reduced any further by analysis even though it is known that the greatest number that cannot be expressed as the sum of two abundant numbers is less than this limit.

Find the sum of all the positive integers which cannot be written as the sum of two abundant numbers.

Considerations and Approach:

Based off of the question we know that we only have to go up to 28123.
We are going to generate all abundant numbers up to 28123 - 12, which is the the smallest abundant number.
After we generate all abundant numbers, we can generate all of the numbers that are sums of two abundant numbers. Then we subtract that from 28123 and voila.

The Code:

```
import math


def generate_primes(upper):
    prime_threshold = upper
    number_list = list(range(2, prime_threshold+1))
    prime_list = []


    current = 0
```

```
    total_iteration = 0
    while current < len(number_list):
        if number_list[current] != -1:
            prime = number_list[current]
            prime_list += [prime]

            increment = current + prime
            while increment < len(number_list):
                number_list[increment] = -1
                increment += prime
                total_iteration += 1

        #print(number_list)
        #print(prime_list)
        current += 1

    return prime_list

def divisor_finder(number, primes):
    upper_limit = number
    #upper_limit = 20023110541 #, good to use to double check. Is [2,2,3,167]
    current_upper_limit = upper_limit
    sqrt_limit = math.sqrt(upper_limit)
    prime_factors = []

    #we are going to hardcode 2 and 3 for looping sake
    i = 0
    while primes[i] < sqrt_limit and i < len(primes):
        while current_upper_limit % primes[i] == 0:
            current_upper_limit = current_upper_limit/primes[i]
            prime_factors += [primes[i]]
        i += 1

    if current_upper_limit != 1:
        prime_factors += [current_upper_limit]

    #print(prime_factors)

    prime_counts = [[prime_factors[0], prime_factors.count(prime_factors[0])]]
```

```
    #print(prime_counts,prime_counts[-1][0],prime_factors[1])
    for i in range(1,len(prime_factors)):
        #print(prime_counts[-1][0],prime_factors[i])
        if  prime_counts[-1][0] != prime_factors[i]:
            prime_counts += [[prime_factors[i], prime_factors.count(prime_factors[i])]]

    #print(prime_counts)

    divisors = [1]
    counters = [x[1] for x in prime_counts]
    #print(counters)
    while sum(counters) > 0:
        new_divisor = 1
        for i in range(len(counters)):
            #print(prime_counts[i][0],counters[i], new_divisor)
            new_divisor *= prime_counts[i][0]**counters[i]
        divisors += [new_divisor]

        counters[0] -= 1
        for i in range(len(counters)):
            if counters[i] == -1:
                counters[i] = prime_counts[i][1]
                if i != len(counters) - 1:
                    counters[i+1] -= 1
            else:
                break

    divisors.sort()
    return divisors[:-1]

upper_limit = 28124

prime_list = generate_primes(upper_limit)

abundant_nums = []
for i in range(12, upper_limit-12):
    divisors = divisor_finder(i, prime_list)
    if sum(divisors) > i:
        #print(i,sum(divisors), divisors)
```

```
        abundant_nums += [i]

print(abundant_nums)

abundant_sum_nums = []
for num1 in abundant_nums:
    for num2 in abundant_nums:
        number = num1 + num2
        if number < upper_limit:
            abundant_sum_nums += [number]
        else:
            break

print(len(abundant_nums))
remove = set(abundant_sum_nums)
keep = [x for x in range(1,upper_limit)]
for number in remove:
    keep.remove(number)

print(len(keep))
```

Euler 0024

The Problem:

What is the millionth lexicographic permutation of the digits 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9?

Considerations and Approach:

Well, this is another problem that Python has a simple built in way...

Generating all of the permutations of the numbers 0-9 is not computationally an issue for itertools, we can then sort those permutations and grab out the millionth permutation and that's our answer.

The Code:

```
import itertools


#Sorry that the next few lines don't look pythonic, I adjusted them for comment clarity
sake
#The line should actually just read
sorted(itertools.permutations(list(range(10))))[999999]

#We sort
sorted(
    #all the permutations
    itertools.permutations(
        #of the numbers 0-9
        list(range(10))
    )
#and then we grab out the millionth permutation
)[999999]
```

Euler 0025

The Problem:

What is the first Fibonacci number with 1000 digits?

Considerations and Approach:

This is somewhat trivial with Python since it can store a 1000 digit number with relative ease, and we don't need to run a super efficient Fibonacci algorithm or store many numbers at all.

The Code:

```
fib_1 = 1
fib_2 = 2
digits_to_find = 1000

#Start at index of 4 because we already have 1,1,2
running_count = 4

#keep going until the next fib digit length is greater than or equal to the digit to find.
while len(str(fib_1 + fib_2)) < digits_to_find:
    fib = fib_1 + fib_2
    fib_1 = fib_2
    fib_2 = fib
    running_count += 1

print(running_count)
```

Euler 0026

The Problem:

Find the value of a,b of n^2 + an + b where |a| < 1000 and |b| <= 1000 where starting from n = 0 and incrementing, there is the largest consecutive chain of primes.

Considerations and Approach:

The way that we can approach this is by incrementing a from -999 to 999 and b from -1000 to 1000.

We only have to generate 1000^2 + 1000*1000 + 1000 as the upper limit of primes that we need to search as well.

We are going to generate all of these primes and as we increment we will first check that the sum of a + b > 1 if it isn't then we are not starting off with a prime. Once we have done this initial check to cull all of the numbers that for sure won't work, then we can check for primality (we could just start the incrementers at the best spot but ehhh...)

To check for primality we generate a list of primes up to the upper limit that we are searching and then convert this list into a dictionary that returns true on a hit. This way, we aren't searching through a list, we are just doing a direct dictionary lookup and verifying that it is indeed prime.

(Without setting up this dictionary lookup this program took minutes and it only got through a = -800 because searching a rather large list using the *in* operator gets expensive near the end of the list. Doing a dictionary only made this whole operation about 1 second, after the prime generation)

Not my most brilliant solution, but fun nonetheless and still efficient enough for the problem space.

The Code:

```
#This generates primes up to a threshold using sieving.
#I've used this sieve generator in many of my problems.
prime_threshold = 2*(1000**2) + 1000
number_list = list(range(2, prime_threshold+1))
prime_list = []
```

```
current = 0
total_iteration = 0
while current < len(number_list):
    if number_list[current] != -1:
        prime = number_list[current]
        prime_list += [prime]

        increment = current + prime
        while increment < len(number_list):
            number_list[increment] = -1
            increment += prime
            total_iteration += 1

        #print(number_list)
        #print(prime_list)
    current += 1

######

#convert the list into a dictionary of {int:True}
#so that .get will return hits or null/false (I don't remember what it does)
quick_prime_list = {}
for prime in prime_list:
    quick_prime_list[prime] = True

#######

#Set up the variables
a = -999
b = -1000
longest_n = [a,b]
longest_n_length = 0
upper_stop_a = 1000
upper_stop_b = 1000 #b is inclusive

#increment a to the upper stop
while a < upper_stop_a:
    #increment b to the upper stop inclusive
```

```
while b <= upper_stop_b:
    n = 0
    #if it is less than 2 then the first number isn't prime...
    if a + b > 1:
        #now we just keep incrementing n while checking the prime list
        while quick_prime_list.get(n**2 + a*n + b):
            n += 1

        #is this the highest n that we have generated???
        if n > longest_n_length:
            longest_n_length = n
            longest_n = [a,b]
    b += 1

#reset the a loop
b = -1000
a += 1

#use as debug to check the status of the a (main) loop
#if a % 10 == 0:
#    print(a)
```

Euler 0028

The Problem:

If you were to build a square starting with 1 in the middle then going right, down, left 2, up 2, right 2, and repeat, all the corner numbers would be diagonals.

On the Project Euler website there is a nice graphic that I'm not putting here.

What is the sum of the numbers on the diagonals in a 1001 by 1001 spiral formed in the same way?

Considerations and Approach:

This problem is bait. We don't have to generate a 2d structure. Instead we can observe the pattern of skipping.
1.3.5.7.9...13...17...21...25.....30

We start with one and add 1. Every four times we add, we add 2 to the skipping distance, and we declare our square 2 larger!

The Code:

```
max_square = 1001


spiral = 1
square_size = 2
total = 1
while square_size + 1 <= max_square:
    for i in range(4):
        spiral += square_size
        total += spiral


    square_size += 2


print(total)
```

Euler 0029

The Problem:

How many distinct terms are in the sequence generated by a^b for a and b being bounded to 2 and 100 inclusive?

Considerations and Approach:

Naively, this is only processing 100*100 numbers, not really much at all.
We can create a python set and then insert every calculation, so it will remove the redundancy.

The Code:

```
lower_limit = 2
upper_limit = 100

#create a set
distinct = set()
#go through the inclusive ranges
for i in range(lower_limit, upper_limit + 1):
    for j in range(lower_limit, upper_limit + 1):
        #do the set addition operator for a^b
        distinct.add(i**j)

#print how many distinct pairs that we created
print(len(distinct))
```

Euler 0030

The Problem:

Surprisingly there are only three numbers that can be written as the sum of fourth powers of their digits: 1634, 8208, 9474
We ignore 1 for the trivial case,lol.

Find the sum of all the numbers that can be written as the sum of fifth powers of their digits.

Considerations and Approach:

We need to find an upper limit first, which we can do by adding 9^5 to a sum until the sum is less than the number of digits.
There is no number that is going to fit these conditions above that upper limit.

Then we increment 1 to the upper limit and find all these multiples

The Code:

```
upper_limit = 9**5
digit_count = 1

#find the upper limit
while len(str(upper_limit))/digit_count > 1:
    upper_limit += 9**5
    digit_count += 1

print(upper_limit, digit_count)

#calculate every number between here and there
digit_fifths = []
for i in range(2, upper_limit):
    current_digit_sum = 0
    for j in [int(x) for x in str(i)]:
```

```
        current_digit_sum += j**5
    if current_digit_sum == i:
        digit_fifths += [i]

print(digit_fifths)
print(sum(digit_fifths))
```

Euler 0032

The Problem:

Pandigital numbers are numbers that have 1-n shown exactly once.
7254 is unusual because 39 * 186 = 7254, which is pandigital through the whole calculation.

What is the sum of all products that can generate these pandigital calculations?

Considerations and Approach:

Seeing that it is 1-9 pandigital, it would be good to skip multiplicand and multipliers that contain 0s.

Assuming a * b = c, we can increment b up until the total digit length is > 9, then we increment a and reset b back to a. When b is reset and a is incremented we can check if total digit length is > 9 and break

The Code:

```
break_threshold = 1000000000 #just something comfy to break

cycles = 0
pands = set()
a = 1
b = 1

#only break if the len of the all the digits is greater than 9 (pigeonhole)
while cycles < break_threshold and len(str(a)) + len(str(b)) + len(str(a*b)) <= 9:
    current_str = str(a) + str(b) +str(a*b)
    #iterate b until the current string length exceeds 9
    while len(current_str) <= 9:
        if len(current_str) == 9:
            #we don't accept 0s, we are looking for 1-9
```

```
            if '0' not in current_str:
                #all digits appear once so the sum of counts should be 9 (9 numbers)
                if sum([current_str.count(x) for x in current_str]) == 9:
                    pands.add(a*b)
        b += 1
        current_str = str(a) + str(b) + str(a*b)

    #increment the cycle
    a += 1
    b = a
    cycles += 1

print(pands, cycles)
print(sum(pands))
```

Euler 0033

The Problem:

The fraction 49/98 is a curious fraction, as an inexperienced mathematician in attempting to simplify it may incorrectly believe that 49/98 = 4/8, which is correct, is obtained by cancelling the 9s.

We shall consider fractions like, 30/50=3/5, to be trivial examples.

There are exactly four non-trivial examples of this type of fraction, less than one in value, and containing two digits in the numerator and denominator.

If the product of these four fractions is given in its lowest common terms, find the value of the denominator.

Considerations and Approach:

Naively... we can iterate through the denominator up to 100, and then iterate through the numerator up to the denominator.

To find the solution we can check every number such that the denominator and numerator have an equal digit, and that the result of the fraction is the same as the digit reduced fractions.

We skip every multiple of 10 over 10 since numerator would make the reduced single digit fraction = 0, which won't ever be the case of the base fraction, or it will be undefined, which is problematic. We can start from 11 every time since we can't digit reduce a single digit number.

The Code:

```
import math


upper_limit = 100
```

```
cycles = 0
non_trivial_den = []
non_trivial_num = []

#denominator and numerator
for denominator in range(11,100):
    for numerator in range(11,denominator):
        if denominator % 10 == 0 or numerator % 10 == 0:
            pass
        else:
            #iterate through the denominator to find numbers that are equivalent
            counter = 0
            for den_index in range(len(str(denominator))):
                for num_index in range(len(str(numerator))):
                    cycles += 1
                    if int(str(numerator)[num_index]) == int(str(denominator)[den_index]):
                        new_num = int(str(numerator)[:num_index] +
str(numerator)[(num_index+1):])
                        new_den = int(str(denominator)[:den_index] +
str(denominator)[(den_index+1):])
                        if numerator/denominator == new_num/new_den:
                            print(denominator, numerator, new_num, new_den)
                            non_trivial_den += [new_den]
                            non_trivial_num += [new_num]

print(cycles, math.prod(non_trivial_den), math.prod(non_trivial_num))
```

Euler 0034

The Problem:

Find the sum of all numbers equal to the sum of their factorials.

Considerations and Approach:

We can find the upper limit by filling a number with 9s and checking that the sum of the digits is greater than the factorial of the number.

After we find the upper limit, we increment through it and check each ones factorial, then take the sum of the digits.

The Code:

```
import math

#find the upper limit
digits = 9
sum_digits = math.factorial(9)
while sum_digits > digits:
    digits *= 10
    digits += 9
    sum_digits += math.factorial(9)

print(len(str(digits)), digits, sum_digits)



### part 2 since we now know the upper limit
upper_limit = 2540160

fact_nums = []
for i in range(3, upper_limit):
    if i == sum([math.factorial(int(x)) for x in str(i)]):
```

```
        fact_nums += [i]


print(fact_nums)
```

Euler 0035

The Problem:

Circular primes are primes that can be caesar'd and still be the prime all the way through.

Find how many circular primes there are below 1 million.

Considerations and Approach:

We are going to generate all the primes under 1000000, then iterate through all of them.
For each prime we will rotate it by it's length - 1 and if every rotation is a prime, then it is a circular prime

The Code:

```
import math


prime_threshold = 1000000
number_list = list(range(2, prime_threshold+1))
prime_list = []

current = 0
total_iteration = 0
while current < len(number_list):
    if number_list[current] != -1:
        prime = number_list[current]
        prime_list += [prime]

        increment = current + prime
        while increment < len(number_list):
            number_list[increment] = -1
            increment += prime
            total_iteration += 1
```

```
        #print(number_list)
        #print(prime_list)
    current += 1

print(len(prime_list))

circ_primes = [2]

prime_list = [prime for prime in prime_list if '0' not in str(prime) and '2' not in
str(prime) and '4' not in str(prime) and '6' not in str(prime) and '8' not in str(prime)]

print(len(prime_list))

iterations = 0
for prime in prime_list:
    iterations += 1
    circular = True
    current = prime
    for i in range(1,int(math.log(prime, 10))+1):
        current = int(str(current)[-1] + str(current)[0:-1])
        #print(prime, current)
        #print(prime, current)
        if current not in prime_list:
            circular = False
            break

    if circular:
        circ_primes += [prime]

    if iterations % 10000 == 0:
        print(iterations/len(prime_list))

#print(circ_primes)
print(len(circ_primes))
```