# Project Euler with SIAM

at ASU West — lead by David Pratt, November 20, 2019

## Introduction

Today we will focus our attention to two challenges. Both challenges lend themselves to extensive planning, which we will do as a group in the case of Project Euler #35 and with the aid of a Numberphile video in the case of Project Euler #37.

We will also continue to use more advanced programming techniques such as stacks and functional programming fundamentals like the `map` and `filter` functions.

## Projet Euler #35

**Circular Primes** *The number 197 is called a circular prime because all rotations of the digits: 197, 971, and 719 are themselves prime. There are thirteen such primes below 100. How many circular primes are there below one million?*

Before we start thinking of an immediate solution to this problem, we should first focus on some properties of circular primes. First off, the most obvious property is that circular primes are themselves prime. Digging a little deeper, we know that prime numbers cannot end in an even digit or the digit 5, so a circular prime cannot contain these digits, else one of its rotations will end in one of these digits.

The latter property alone eliminates a huge quantity of numbers from our consideration. But how do we ignore numbers that contain these digits? We have two options:

(i) We can iterate over all the numbers, but whenever we encounter a number that contains these *problem* digits, we simply skip the number.

(ii) We can generate all possible numbers less than one million that do not contain these digits.

Intuition tells us that generating numbers may be more efficient than iterating over one million numbers and filtering out the unwanted numbers. In fact, we can find just how many numbers have the property of not containing any even digits or the digit 5 that are less than one million. Below 10, there are 4 such numbers, namely 1, 3, 7, and 9. Below 100, there are 20 such numbers. Mathematically, we can see this using the Cartesian Product.

Let $D = \{1, 3, 7, 9\}$. The numbers less than 100 with two digits with this property are all present in $D \times D$ (if we take each ordered pair to represent a number in the sense we would expect). Clearly, there are $4^2 = 16$ such numbers. So there are $4 + 16 = 20$ such numbers less than 100.

Below one million, there are $4 + 4^2 + 4^3 + 4^4 + 4^5 + 4^6 = 5{,}460$ numbers with the desired property. All this is to say that generating these numbers is far more efficient than iterating over one million numbers and testing for the property.

How do we generate these numbers, then? The Cartesian Product! Luckily for us, the Cartesian Product is implemented in the standard library of Python in `itertools.product`.

For example, the statement `list(product([1, 3, 7, 9], repeat=2))` would yield the list of tuples `[(1, 1), (1, 3), ..., (9, 7), (9, 9)]`. Note that we called `list()` on the `product` function. This is because `product` is a generator function, meaning it doesn't actually **`return`** any values, rather it **`yield`**s values.

We still haven't generated the numbers we want (in fact, we haven't generated any numbers at all). We need to implement a function `tuple_to_int(t)` that converts a tuple to an integer. To do this, think about how numbers are constructed in base ten.

The number 2718 can be written as $2718 = 2 \cdot 10^3 + 7 \cdot 10^2 + 1 \cdot 10^1 + 8 \cdot 10^0$. We can express this in words as "the sum of the product of the digits with the powers of 10 from 0 to 3." In Python, we can express it in almost the same way.

```
def tuple_to_int(t):
    """
    Converts a tuple of integers to an integer.

    Example:
        >>> tuple_to_int( (2, 7, 1, 8) )
        2718
    """
    return sum(digit * pow(10, p) for p, digit in enumerate(reversed(t)))
```

Fantastic! Now we can generate all $k$ digit numbers with our special property with the single line of code `map(tuple_to_int, product([1,3,7,9], repeat=k))`.

Now we just need to implement a function `rotations(n)` that gives all rotations of an integer $n$. To do that, we might want to think about how to rotate a number a single time.

Let's try to rotate the number $n = 197$ to $n' = 719$. We can use floor division by 10 to get the head $h$, or the first digits, of the number. To get the tail $t$, or last digit, we can simply take $n$ modulo 10. In Python, we can get both in a single line with the assignment `head, tail = divmod(n, 10)`. Now, we see that

$n' = t10^2 + h$. All together, the function may look like this:

```
def rotate(n):
    """
    Returns a single back-to-front rotation of n.

    Example:
        >>> rotate(197)
        719
    """
    head, tail = divmod(n, 10)
    return tail * pow(10, digits(head)) + head
```

Writing a function to generate all rotations is all-of-the-sudden not such a big task. We just have to rotate an integer $n$ a few times to get all rotations of $n$.

```
def rotations(n):
    """
    Generates all unique rotations of a number n.

    Example:
        >>> list(rotations(197))
        [197, 719, 971]
    """
    for _ in range(digits(n)):
        yield n
        n = rotate(n)
```

All that is left to do is implement the algorithm that finds circular primes. Since we are only considering numbers that contain the digits 1, 3, 7, and 9, we initialize a set of circular primes with 2 and 5, since they are considered to be circular primes. Then, we iterate over the numbers from 1 to 6 (inclusive) to generate all of the numbers that only contain 1, 3, 7 or 9. For each of the numbers we generate, we check if it has not already been found to be a circular prime and that it is prime itself. If it passes that test, we generate all of its rotations, test them all for primality, and if that test passes, we update the set of circular primes.

```
def circular_primes_range(p):
    """
    Calculates all circular primes less than a given power of 10.
    Returns a set of all such primes.

    Example:
        >>> circular_primes_range(6)
        {2, 3, 5, ..., 999331}
    """
    # Circular primes cannot contain even numbers or the digit 5, so the
    # only digits allowable in a circular prime are:
    DIGITS = [1, 3, 7, 9]
```

```
    # We make an exception for the single digit circular primes, 2 and 5
    # It is also imperative we use a set to store circular primes due to
    # the fact that we will be performing lookups repeatedly.
    circular_primes = {2, 5}

    # We look for circular primes less than 10**p, meaning numbers with
    # a digit count of 1 through p (inclusive).
    for num_digits in range(1, p + 1):
        # The product() function is an implementation of the Cartesian
        # Product. In our case, it will produce...
        #
        # product(DIGITS, repeat=1) -> (1,),      (3,),      ...
        # product(DIGITS, repeat=2) -> (1, 1),    (1, 3),    ...
        # product(DIGITS, repeat=3) -> (1, 1, 1), (1, 1, 3), ...
        # ... and so on ...
        #
        # Since product() produces tuples, we map each of the tuples to
        # an integer by using the map function.
        nums = map(tuple_to_int, product(DIGITS, repeat=num_digits))

        for n in nums:
            # If n has not already been found to be a circular prime
            # and is prime itself...
            if n not in circular_primes and isprime(n):
                # Collect the rotations of n in a set rots. If all of
                # the rotations are prime, add them all to the set of
                # circular primes.
                rots = set(rotations(n))
                if all(map(isprime, rots)):
                    circular_primes.update(rots)

    return circular_primes
```

I have implemented the function with a parameter so those who are curious can find larger circular primes... *if they dare...*

## Project Euler #37

**Truncatable Primes** *The number 3797 has an interesting property. Being prime itself, it is possible to continuously remove digits from left to right, and remain prime at each stage: 3797, 797, 97, and 7. Similarly we can work from right to left: 3797, 379, 37, and 3. Find the sum of the only eleven primes that are truncatable from both left to right and right to left. (The numbers 2, 3, 5, and 7 are not considered to be truncatable primes.)*

If you haven't already, now would be a good time to watch the Numberphile video titled *357686312646216567629137*. At timestamp 1:56, Dr. James Grime walks through the beginning stages of an algorithm to find all left truncatable primes. Dr. Grime begins by listing all single digit primes, and then describes the process of appending digits to the left side of the prime, creating nine new numbers. Any of these nine numbers that remain prime must be left truncatable primes. We can repeat the algorithm until we have exhausted all paths.

As it happens, there are thousands of left truncatable primes whereas there are roughly 80 right truncatable primes. So in the context of this challenge, we should aim to generate right truncatable primes, find those of which are left truncatable, and then sum them. Let's build an algorithm to find all right truncatable primes.

The idea will be to keep a stack[1] in which only right truncatable primes are present. Right truncatable primes will be popped off the stack so we can generate numbers that are candidates for being right truncatable. Those that are prime (and therefore right truncatable) will be pushed onto the stack and collected separately. This process will continue until the stack is empty. Below is a visual walkthrough of the algorithm.

We will begin with all single digit primes in a stack.

$$\begin{bmatrix} 2 & 3 & 5 & 7 \end{bmatrix}$$

Pop the stack to obtain a right truncatable prime (in this case, 7) and generate possible right truncatable prime candidates by appending the digits 1, 3, 7, and 9 to the number.

$$\overbrace{\phantom{71\quad 73\quad 77\quad 79}}^{7}$$
$$71 \quad 73 \quad 77 \quad 79$$

Push only the generated numbers that are prime onto the stack, and collect them in a set elsewhere. The stack now becomes:

$$\begin{bmatrix} 2 & 3 & 5 & 71 & 73 & 79 \end{bmatrix}$$

Repeat this process until the stack is empty.

Before we write the Python code, let's prove that this algorithm actually produces all right truncatable primes.

*Proof.* Assume that the algorithm does not produce all right truncatabe primes. Let $A$ be the set of all right truncatable primes produced by the algorithm and let $E$ be the (non empty) set of right truncatable primes that elude the algorithm.

[1] A stack is an incredibly useful data structure similar to a list that supports access from exactly one end through pop and push operations.

Take $x = \min E$. Since $x$ is right truncatable, its right truncation $x' = \lfloor x/10 \rfloor$ is a right truncatable prime. Furthermore, because $x' < x$, we see that $x' \in A$.

But since $x'$ was generated by the algorithm, $x$ would be generated as well. Contradiciton! Fake news! Absurd! □

Now that we are convinced our algorithm will produce an exhuastive list of right truncatable primes, let's translate it into Python code. Note that we yield the right truncatable primes as we find them, as opposed to collecting them in a set.

```
def right_truncatable_primes():
    """
    Generates all right truncatable prime numbers.
    Assumes that single-digit primes are not truncatable.
    """

    # Prime numbers with more than one digit can only end in a 1, 3, 7,
    # or 9.
    DIGITS = [1, 3, 7, 9]

    # Begin initially with the single digit prime numbers. From these we
    # can construct all right truncatable primes.
    truncatable_primes = [2, 3, 5, 7]

    # while the truncatable_primes stack is non-empty:
    while truncatable_primes:
        prime = truncatable_primes.pop()
        for d in DIGITS:
            # A possible right truncatable prime is any right
            # truncatable prime concatenated with a digit 1, 3, 7, or 9
            candidate = 10 * prime + d

            # If the candidate is prime, then it is a right truncatable
            # prime, so we add it to the truncatable_primes stack and
            # yield it.
            if isprime(candidate):
                truncatable_primes.append(candidate)
                yield candidate
```

All that remains is to filter for the left truncatable primes, so we need to implement a function that checks if a number is a left truncatable prime. To do this, we will proceed similarly to how we generated all rotations of a number in the previous challenge.

First, we will implement a function to return a single left truncation of an integer $n$.

```
def truncate_left(n):
    """
    Returns a single left truncation of n.

    Example:
        >>> truncate_left(9137)
        137
    """
    return n % pow(10, digits(n) - 1)
```

Now, use this function repeatedly to obtain all left truncates of an integer $n$.

```
def left_truncates(n):
    """
    Generates all left truncates of an integer n.

    Example:
        >>> list(left_truncates(9137))
        [9137, 137, 37, 7]
    """
    for _ in range(digits(n)):
        yield n
        n = truncate_left(n)
```

Now that we have generated all left truncates of an integer, we can easily find if they are all prime.

```
def is_left_truncatable(n):
    """
    Determines whether n is a left truncatable prime number.

    Example:
        >>> is_left_truncatable(9137)
        True
    """
    return all(map(isprime, left_truncates(n)))
```

With one simple line of code, we will have the answer to the challenge.

```
    sum(filter(is_left_truncatable, right_truncatable_primes()))
```