Explanation to the first thirty problems from Project Euler with Python 3

Max Halford

Project Euler is a good way to learn basic number theory, to get your imagination going and to learn a new programming language. If you can solve the first hundred problems then you can solve any problem, as long as you keep being curious and you use your imagination, personally I decided to work on other styles of projects, there isn't just number theory out there! However solving number theory problems is a good way to learn a programming language : you need to be rigorous and tidy. Most solutions require smart algorithms and not brute force approaches. Google is your best friend when you know what to do but you don't know how to write it, or when you don't understand the code you're reading. I will not babysit the reader but to the contrary assume that he knows how to google "Python <insert command>", I am not being harsh : knowing how to find documentation when coding is of the utmost importance because you mostly have to teach yourself notions, you have to be self-educated. I would recommend approaching challenging problems with pen and paper and maybe some mathematical research. The association between human ingenuity and the computational power of our machines can produce wonderful results, as long as one doesn't lean on the latter. Python provides many coding styles and paradigms, googling "PEP 8" and "Google Python Style Guide" will make pick up good habits early on. Keep your code simple, it has to be readable by everyone. Assign comprehensible names to your variables and parameters. Don't reinvent the wheel, the Python community is very large and modules have been written for a lot of things, use them. Internet isn't always right, for example one liners are not a good thing : they are difficult to read afterwards. In a perfect world reading your code should feel like reading a book, keep that in mind. Don't be frightened to go on internetand find the solutions to problems, it's actually counterproductive to search for answers for too long, time is precious and there are too many things to cover. All the code is available here:

https://github.com/MaxHalford/Project-Euler

Contents

1 Multiples of 3 and 5 **4**

2 Even Fibonacci numbers **4**

3 Largest prime factor **4**

4 Largest palindrome product **5**

5 Smallest multiple **5**

6 Sum square difference **5**

7 10001st prime **6**

8 Largest product in a series **6**

9 Special Pythagorean triplet **6**

10 Summation of primes **7**

11 Largest product in a grid **7**

12 Highly divisible triangular number **8**

13 Large sum **9**

14 Longest Collatz sequence **9**

15 Lattice paths **10**

16 Power digit sum **10**

17 Number letter counts **11**

18 Maximum path sum I **11**

19 Counting Sundays **12**

20 Factorial digit sum **12**

21 Amicable numbers **12**

22 Names scores **13**

23 Non-abundant sums **13**

24 Lexicographic permutations **14**

25 1000-digit Fibonacci number **14**

26 Reciprocal cycles **15**

27 Quadratic primes **15**

28 Number spiral diagonals **16**

29 Distinct powers **16**

30 Digit fifth powers **17**

1 Multiples of 3 and 5

A question that often comes up with Python is "What is the best way to go through a list?". Problematically there are many answers and it all comes down to personal preferences. The two main choices are Map-Reduce-Filter-Lambda `http://www.python-course.eu/lambda.php` and list comprehensions `http://www.python-course.eu/list_comprehension.php`. I would suggest reading up on both approaches, it is always good to know what tools are at your disposal without having to be a black belt at them. I mostly use list comprehensions, they are, in my opinion, more comprehensible and flexible. The problem is straightforward with a list comprehension : build a list bounded by 1 and 999 with elements being divible by 3 or 5 and sum up all the elements.

2 Even Fibonacci numbers

When you read a problem for the first time do as much research as you need on the topics of the problem. In this case Fibonacci numbers are relatively famous but will find some rather less famous concepts later that will require some looking around. After some pondering one notices that a number in the Fibonacci sequence is only defined by the two numbers preceding it. What results from this observation is that it is useless to store the sequence, but instead we should go through it and pick the numbers we want (in this case they have to be even). We use the property of the sequence in the following algorithm :

- Let α and β be adjacent numbers in the sequence.
- β' becomes the following number in the sequence : $\alpha + \beta$.
- α' becomes β.

3 Largest prime factor

It is a well known theorem that any number can be written as the product of prime numbers. Hence any number divided by all its factors successively will return 1. So what we can do is, for a given integer n, go through every integer k inferior to it and superior to 1 and check if $n \bmod k = 0$. If it isn't then increment k and try again. When it is then k is a factor of n, so we store it and start over the same process for n/k. The neat thing with this algorithm is that you do not have to check if k is prime. Indeed k will be checked before $2k, 3k, 4k$ *etcetera*.

4 Largest palindrome product

The easiest way to check if a number, or any string for that matter, is a palindrome, is to compare it with its reverse. I would suggest reading `http://forums.udacity.com/questions/2017002/python-101-unit-1-understanding-indices-and-slicing` to learn useful dodges instead of reinventing the wheel. Now that we have a function to check if a number is palindrome, we need to apply it on every product of two 3-digit numbers in the descending order (because we are looking for the biggest palindrome), hence the iteration through $\{999, \ldots, 100\} \times \{999, \ldots, 100\}$. Notice how easily a two dimensional matrix is defined in one line with a list comprehension.

5 Smallest multiple

Recursion is a concept that can not be avoided when programming, I would suggest googling it if you are not familiar with it. It is easy enough to check for a given integer n from 1 till k (in our case 20), we simply go through the list and check the divisibility. Now we have to find an integer that satisfies the previous algorithm for $k = 20$. A brute force approach is to iterate every integer one by one, but we quickly realize that the algorithm takes for ever. It takes some insight to notice that the integer n that verifies the algorithm for k is a factor of the integer m, hence when looking for m we can increment by n instead of 1. However efficient this insight is, it is not intuitive, try some cases on a piece of paper. An example that comes to mind is this : 6 is divisible by 1, 2 and 3. 24 is divisible by 1, 2, 3 and 4. 6 is a factor of 24 hence we only look for the candidates 12, 18 and 24 when searching for the smallest integer divisible by 1, 2, 3 and 4.

6 Sum square difference

I'll be honest, if you don't konw how to do this problem, it is either than you need to learn the basics of programming or that you're not motivated. This is good opportunity to learn some more list comprehension synthax. The solution to this problem is straightforward, indeed we only have to iterate through a list of integers and do some basic operations on them.

7 10001st prime

I would strongly recommend reading up on prime number theory, it is a core element of number theory and it is not an easy concept to deal with in computer science (actually that's the reason why it is used in cryptography). Indeed, the algorithms used to deal with primes require a lot of power, even without brute force. Again, there are many to solve this problem, the upper bound (10001) is relatively low so we can use an unsophisticated algorithm. To check if an integer is prime, one can go through every integer in $\{2, \ldots, \sqrt{n}\}$ and check if it divides n (going above $\sqrt{n}$ is pointless because you've implicitly checked them when going through the smaller integers). Now that we have a tool to check if an integer is a prime we can through the uneven integers (even numbers apart from 2 are not prime) and use the tool. Once we have 10001 primes we return the last one. On a sidenote, storing the primes is overkill, it's fine for 10001 primes but you might want to change the script (just like in Problem 2) to only store the latest prime.

8 Largest product in a series

This is a good exercice to master ranges in Python (the last element of the range not being included is an easy concept to forget at first). First of all we have to copy/paste the number into a Python script and edit it so that it's considered a prime. In my solution I quoted it and made it readible by adding '/' at the end of the lines (Python thinks it will be one single continuous line). You could also use `list(map(int,str(n)))` to tranform n into a list. The easiest way to find a maximum is to iterate through a list and compare to an initial value, if we find a bigger "thing" we replace the initial value by the "thing". In our case the "thing" is a thirteen elements product. Thus we need two loops, one going through every element i, the other going from i to $i + 12$.

9 Special Pythagorean triplet

It is always important to know just how many steps are needed to get to the answer. In this case there are two : firstly imposing two criterias on an integer triplet and secondly computing pythagorean triplets. However, after a bit of coding and reflexion we should notice : if we have a and b, we automatically have c because the three have to sum to 1000! Also, the Pythagoras theorem ($a^2 + b^2 = c^2$) imposes on c to be bigger than a and b

(think of a triangle and its sides and the previous statement will seem clear). In mathematical terms the first observation says that $c = 1000 - a - b$. The second says that a and b can only be as big as 500 (half a thousand). Let's prove this statement *ad absurdum*, if $a > 500$ and $c > a$, then $a + c > 1000$, which we don't want. Finally we can notice that a and b are interchangeable (by commutativity of the sum), thus we can iterate through $\{a, \ldots, 500\}$ for b to avoid repeating triplets. Now that we have understood all this the coding is simple, we go through appropriate triplets and check if the triplets verify $a^2 + b^2 = c^2$.

10 Summation of primes

This problem is the perfect example for using the sieve of Eratosthenes, which is an efficient algorithm to find all the prime numbers up to a given bound. It uses the following intuition : if we know for certain that k is prime, then all the multiples of k are not prime. We can use this property in the following way :

- Take a list of n consecutive booleans set to `True` (2 million in our case).
- Go through every element.
- If it is $True$ (prime) then mark all its multiples to $False$.
- If it is $False$ (not prime) then check the next element.

Finally we end up with of a list of $True$ and $Falses$, sum up the integers that are $True$.

11 Largest product in a grid

Some problems don't require elegant dodges but simply clean algorithms, this is one of them. First of all there are four possible products : a column ($\downarrow$), a line ($\rightarrow$) and two diagonals ($\searrow$ and $\nearrow$). I didn't add the symbols to be pedantic : the way they point is the way I multiplied the cells of the grid, just "go the way" that seems most natural to you. What we want to do is straightforward, we compute every kind of product for each cell of the grid. However before doing the previous the given grid has to be inserted into a two-dimensional array, first by copy/pasting and converting to string the given grid then by adding every line to an array. We also have to be careful not to try and compute inapropriate products, for example a line product on the last cell of a line of the grid. I found that getting the array into place

was harder than finding the largest product, mastering string operations is crucial. In my solution there are three string operations :

- `.strip()` removes the implicit carriage returns at the end of every line.
- `.splitlines()` converts a multiline string into individual strings.
- `.split()` separates a string into a list of strings based on a given separator.

12 Highly divisible triangular number

It is fairly obvious that the nth triangular number is the sum of an arithmetic sequence : $n(n+1)/2$. We can make two observations : one and only one of n and $n+1$ is divisible by 2 and they don't share any prime factors. Thus we can write :

$$n = \prod_{i=1}^{s} p_i^{j_i} \text{ and } n+1 = \prod_{i=1}^{t} q_i^{k_i} \text{ where } p \text{ and } q \text{ are primes.}$$

Thus the number of factors of a triangle number is:

$$(j_1)(j_2+1)\dots(j_s+1)(k_1+1)(k_2+1)\dots(k_t+1) = j_1 \prod_{i=2}^{s}(j_i+1)\prod_{i=1}^{t}(k_i+1)$$

I realize that the previous equation is a lot to take in so I'll explain. First of all, if an integer n has s prime factors p, then any product of these primes (each to any power from 0 to j) is a factor of n (this is the most important insight to understand). The previous statement induces that the number of factors of an integer is $(j_1+1)(j_2+1)\dots(j_s+1)$. The reason for which there is a "+ 1" is that we have to include 0 to the powers of the primes. However a triangular number is equal to $n(n+1)/2$, not to $n(n+1)$, which means that we have to neglect a power of two in the factorization of n or $n+1$ (depending on which is even. This explains why the first element of the product is (j_1) and not (j_1+1).
Now that we have the theory we can put it into code. Firstly we need an algorithm to give the number of divisors of a given integer n. My algorithm is a bit complicated but it exactly translates what was said above. We begin by dividing by 2 n until it is odd and count how many times we did it (j or k in the previous paragraph). We then divide n (which is now odd) by its odd prime factors until it equals 1, in this process we also count how many times we can divide n by the odd number. The code `divisors = divisors * (count + 1)` is our insight translated to code : we multiply the number of divisors we have by the number of times we can divide n

by one of its prime factors. Next we have to iterate through every integer until `numDivisors(n)` $\times$ `numDivisors(n+1)` is over 500. Finally our answer is simply $n(n+1)/2$.
This is the first problem where we looked at our problem under a different angle. We thought about what equation represents a triangular numbers and from there worked on the equation, which is a subtle difference and is often the case when solving hard problems.

13 Large sum

We can use the code used in Problem 11. First of all we copy/paste the given number, strip the carriage returns. Then we convert every line to an integer which we append to an array. Finally, after summing the array and converting the sum (an integer) to string, we can simply get the last 10 digits by *slicing* it. Slicing in Python is an important tool to master, it makes string operations very simple, if you don't understand it by now go back to the link in Problem 4.

14 Longest Collatz sequence

The Collatz Problem is a fascinating, unproved, conjecture. Let's say that $C(n)$ returns a sequence of integers, beginning with n and ending with 1, every integer following the rules given by the Collatz sequence :

- n becomes $n/2$ if n is even.
- n becomes $3n+1$ if n is odd.

We are looking for the biggest $C(n)$ (denoted $|C(n)|$) for n in $\{1, \ldots, 1000000\}$. Every n returns a unique Collatz sequence so we have to check each $C(n)$. The first thing that comes to mind is to compute the length of each $C(n)$ for each n, however this takes a *very* long time. Consider the following insight : the first element of $C(10)$ is 5, yet we have already computed $C(5)$, thus $|C(10)| = |C(5)| + 1$. This is wonderful, when computing $C(n)$ we can stop once it returns an element k for which $C(k)$ has already been computed. We then have $|C(n)| = |C(k)| + l$ where l is number of elements between n and k. This is principle is called *memoization*, it avoid repeating steps already computed. It finds applications in everything that is *recursive*. For example when recursively computing `factorial(n)` we can stop if we computed `factorial(n-k)`, indeed $n! = \frac{n!}{(n-k)!} \times (n-k)!$.
Back to the problem, the memoization doesn't change much to the brute

force algorithm described previously. Only now before changing n we have to if n is in the keys of a dictionary (also known as a hash) instantiated formerly. If it is then we can stop and we have the length of the Collatz sequence beginning with n. If it isn't then we keep going. In any case when the length of the sequence is found we append it to the dictionary so that we don't have to the computation again if we find n in another sequence.

15 Lattice paths

There are two ways of solving this problem. On the one hand we could use a brute force recursive algorithm by computing the number of routes for subgrids and work our way up to the 20×20 grid. On the other hand a bit of peripheral vision tells us that this is a famous problem in combinatorics. Take a grid of size $n \times m$, it makes sense that to go from top-left to bottom-right you have to go down n times and right m times. The real insight is to notice that if all the movements in one direction are imposed, then there is no more liberty for the movements in the other direction : they are predetermined. But this is also with a mix of both movements, as long as the mix involves half of all the movements necessary to go from one point to another. Indeed on a 20×20 grid, if 5 movements downward and 5 movements to the right are imposed, then the other 10 are predetermined (try it out for yourself to convince yourself). Basically we are looking for how many ways there are to arrange $(n+m)/2$ among $n+m$ possible movements (round $(n+m)/2$ up or down if it is not whole, it doesn't matter). In combinatorics this is denoted $\binom{n+m}{(n+m)/2}$ which is $\frac{(n+m)!}{((n+m)/2)((n+m)/2)}$. In our case the grid is square of size 20, which makes things cleaner (no problems with $(n+m)/2$ being odd). The answer is simply $\frac{40!}{20!20!}$. This solution is faster than the first way of solving the problem, however some may argue that it not the computer science way of doing it. I agree, however having a big toolkit will spare you many problems, thus having a peripheral vision is good. You don't have to be a black belt at everything, that's impossible, however knowing what tools exist is feasible. If you know what tool to use in what case or you know in what domain to look into then you can do some research and solve problems elegantly.

16 Power digit sum

I wouldn't say there is anything nifty with this problem. One simply has to convert 2^{1000} to string and iterate through its digits, simple as.

17 Number letter counts

Now this is the kind of problem where modular arithmetic becomes handy. Say you want to strip a number n it of its digits above the units digit, well the clean way to do so it to calculate $n \bmod 10$. To get the tens digit calculate $\frac{(n \bmod 100)-(n \bmod 10)}{10}$, indeed for digits above the units digit you have to remove all the digits below it if you want it on its own, you also need to divide it by its amplitude to remove all the zeroes that you will then have. Once we have mastered this the problem becomes quite simple. We iterate from in $\{1..999\}$ and do some modular operations to get the units, tens and hundreds digits. We now have to use common sense and check some conditions. Is there any hundreds digit? Is there an "and" in the number? The first question is straightforward. For the second one only has to check if the number contains a tens or a units digit. The last question is a bit trickier and is related to the nature of the english language. Indeed numbers from eleven to nineteen are special because they each have a unique name, all the others are preceded by the name of the amplitude of the tens digit (twenty, thirty, *etcetera*). Finally we have to add the number of letters of 1000, computing it in the loop wouldn't have made sense since it was the only number with a thousands digit.

18 Maximum path sum I

If you remember the labyrinth games you played when you were young and the fastest way to win at them you'll quickly solve this problem. The fastest way to win wasn't to try out every path from the start, but rather to follow the path from the end and see what path lead to it from the start. The same idea applies here, we won't try out every path from the path but go from bottom to top. Every node has two child nodes, we can just add the value of the biggest child node to the value of the node we're looking at. The last row doesn't have any child node so we can start at the penultimate row. For the algorithm itself to work we first have to put the tree in shape, I used a list comprehension to convert the copy/pasted string form of the tree to a list of lists. Once we an array we can iterate through it starting from the penultimate row and working our way up with the use of `range(len(T) - 2, -1, -1)`. The first element of the range is the penultimate row index, the second if the first row (we put `-1` because `range()` is not inclusive on the last index), the third simply gives the way to iterate through the range. On each row we increment each element by its highest child node.

19 Counting Sundays

Of the first problems this is the one I had the most fun doing. Just like Problem 17 it connects with the real world. What I mean is that calendars and languages are human inventions, they are not intrinsic to our universe, yet we can still manage to deal with them by adapting and creating mathematical tools. If you're curious there is a lot of content on the topic of calendar and mathematics, you could start by reading `http://www.wikipedia.com/en/Determination_of_the_day_of_the_week`. First of all we can notice that a 7 (numbers of days in a week) doesn't divide 365 nor 366 (leap year), thus the first day of a year will not be the same as the year before. In fact we can predict which day it will be next with modular arithmetics. Indeed $365 \equiv 1 \pmod 7$ and $366 \equiv 2 \pmod 7$. Now we know that a leap year starting on a Monday will be followed by a year beginning by a Wednesday. We can use the same method to calculate the first day of next month, for example if August begins with a Monday then September will begin with a Thursday because $31 \equiv 3 \pmod 7$. Last and not least we need a function to check for leap years. If a year is divisible by 4 it is a leap year, except if it divisible by 100 and not by 400. We now have all the tools we need. Let's sumply iterate through each year and each and modify a flag value with modular arithmetic, each time the flag value becomes a Sunday we increment a counter.

20 Factorial digit sum

If you managed to solve Problem 16 this one will not be an issue providing you know how to compute the factorial of a number.

21 Amicable numbers

I didn't anything nifty for this solution, it's plain brute force. However the time to solve the problem is sub ten seconds so it's not a problem. We only need a function to return the sum of the divisors of a number. Once we do we can iterate through $\{1..10000\}$, compute $sumDivisors(n)$ and compare it to $sumDivisors(sumDivisors(n))$. In others we compute the sum of the divisors of the sum of the divisors of a given number. I imagine that you could faster than y script, maybe with a better $sumDivisors()$ function.

22 Names scores

Like many problems this one is all about successive simple steps: open the file, put the names in a list, sort them, compute scores. The *os* module in Python enables us to choose a directory, for example I put a relative path in the code. I'm not sure if this is true for every operating system but on Ubuntu a script will look for files in the directory it is placed in, so that we don't have to it where to look. Once you have figured this out simple use `open()` and `read()` to get the names as a long string. I then used a list comprehension to split the names at commas whilst stripping the names of the brackets surrounding them, but you can do it another way if it doesn't seem natural to you. I think that list comprehensions are powerful and succint, providing you have gotten used to them. The computer doesn't know what value we have assigned to each letter of the alphabet so we have to tell him. The best way to do so it to create a dictionary containing these values. Now we can iterate through every name, iterate through every letter and multiply the score of a name by its position in the names list (by checking where we are in the loop).

23 Non-abundant sums

I believe the best way to approach this problem is to proceed in two steps. First we need to find all the abundant number under 28123. Then we can find every sum of abundant numbers possible and to the following: if a sum of abundant numbers is inferior or equal 28123 then we store it into a sum called S, then we can calculate $S' = \sum_{i=1}^{28123} i$ (using the arithmetic sum formula) and $S' - S$ will be our answer. In other words, we are using set theory: we calculate the sum of all the numbers below 28124, then we calculate the sum of the numbers below 28124 that can be written as the sum of two abundant numbers and calculate the difference between both sums. This explains why we only compute abundant numbers below 28123: the sum of two abundant numbers above 28123 is already known to exist so we can discard them. The `divisors(n)` function I used is more efficient then the one in Problem 21. It uses the fact that if we know that k divides n, then n/k divides n, provided $k \neq n/k$.

24 Lexicographic permutations

We have to do some thinking before delving into this one. Let us look at the example given at `https://projecteuler.net/problem=24`. Let us say we are looking for the 5^{th} permutation, well that would be 201. You can think of each pair of permutations as three families. The first one being $(012, 021)$, the second $(102, 120)$ and the last $(201, 210)$. We can see the 5^{th} permutation as the 1^{st} element of the 3^{rd} family. Let that sink in if it has not already. To find this mathematically is equivalent to perfoming a modular operation: $\lfloor \frac{objective}{(n-1)!} \rfloor$ where n is the number of elements considered in the permutations (here it is 3). Let us see how it works out for the previous example: $\lfloor \frac{5}{(3-1)!} \rfloor = 2$, which means that our objective is in the third family $(2 + 1 = 3)$, in other the first two families contain the first four permutations. We can keep going and apply the same operation to the remainder of the euclidian division we performed (1). However now we know that the first of the permutation is fixed because we know what family it is in (the one that starts with 2), thus we decrement n: $\lfloor \frac{1}{1!} \rfloor = 1$. This does not mean that our objective is the first element of the second family, it means that our objective is in the first subfamily of the family (and not the second because $1 \equiv 0 \pmod 1$, which was not the case in the first part of the loop). However the subfamily (201) is only one element, thus it is the answer. If it was not on its own we would have to keep going. I will recapitulate by describing the algorithm. First we compute the euclidian division of our initial objective by the permutations of the numbers of element n minus 1 ($n!$ is obviously bigger then any position we could be looking for). We now the first digit of the permutation we are looking for because $(n - 1)!$ fits that many times in our objective. We now have a remainder which is the number of elements we have to look into the same way. In other words it is our new objective. We know part of where our objective lies and we have to delve into this family of permutations until we find a single possibility, the final leaf on the tree.

25 1000-digit Fibonacci number

Let's use the algorithm from Problem 2 to generate Fibonacci numbers whilst incrementing a counter until we find one that is composed of at least a thousand digits.

26 Reciprocal cycles

From my research I have only found one way to solve this problem: *Fermat's little theorem* (`http://www.wikiwand.com/en/Fermats_little_theorem`). If you are interested you can do some research on the subject. The application of the theorem we interested in is the following: $\frac{1}{d}$ has a cycle of n digits if $10^n - 1 \equiv 0 \pmod{d}$. Another property that can be deduced with this theorem is for any prime p, $\frac{1}{p}$ has at most a recurring cycle of $p - 1$ digits. Honestly I don't like these kind of problems. By nature this problem can't be solved with brute force algorithms because we don't know what maximum length a cycle can be. With mathematics we can find a boundary. However not everyone can master complicated mathematics like these, what is important is to recognize when a problem can't be solved with classical methods and look for theory. *Know the tools at your disposal.* We now have two bits of theory that can be plugged into a simple loop. First of all we collect all the prime numbers inferior to 1000. Then we iterate through them in reversed order (bigger primes will producing bigger recurring cycles). Now we want to find the number of digits in the reccurring cycle for each prime p, for this we simply check increment a counter n until we find $10^n - 1 \equiv 0 \pmod{p}$. Finally we check if the recurring cycle has $p - 1$ digits, if it doesn't we do the same thing for a different thing, it it does it is the answer.

27 Quadratic primes

Really there is nothing complex with this problem. If we go from basis that we can't induce properties for a polynomial based on another then we simply have to iterate through each possibility. First of all we need a function to test if a number is prime, we already have that in our toolkit, for example from Problem 10. Next we need to check how many primes a polynomial can generate. For this we create a function that takes as arguments the coefficient a and b and keeps incrementing n until it returns a composite number. Now for the candidates we could try every possibility, but that would be four million, which is unreasonable. By looking at the polynomials we can notice that if $n = 0$ then what is left is b, thus b has to be prime, which also means it has to be positive. The algorithm will run much faster now. Some people will point out the fact that a has to be odd and bounded by $-b$ and b, the speedup is not essential. As a bonus I included a plot in the code and you can notice that indeed a lot of couples (a, b) generate only one prime because there are no constraints on a.

28 Number spiral diagonals

This problem is good for people who like toying with numbers. Being one of them I won't give a brute force solution but an elegant one. I would recommend taking a piece of paper, a pen and looking at the example grid from the website. First of all we can notice that along the diagonals there are always four numbers, the top right one being the biggest, the top left the second biggest, the bottom left the third biggest and the bottom right the smallest. It is quite obvious that the sum of the left side numbers are equal to the sum of the right side numbers. Hence we can calculate the right side numbers and multiply their sum by two. What we need is a formula that gives the n^{th} top right number and another formula for the n^{th} bottom right number. Let's start with the top right numbers. Our intuition should tell us that these numbers are simply the squares of odd numbers. The n^{th} odd number is given by $2n-1$, thus the n^{th} top right number is $(2n-1)$. Now for the bottom right corner. It takes a bit more time to notice there is a link between each corner, indeed that's we decided to only calculate only two corners. The corner values go progressively down in the anticlockwise way. For example on the first square, the top right corner is equal to 9, is goes down to 7, then 5, and finally 3. On the second square it's 25, 21, 17, 13. The conjecture that follows is that the reduction for the n^{th} from corner to corner is $2(n-1)$. From this conjecture we know that the bottom right corner is equal to the top right corner minus three times the reduction, hence $6(n-1)$. To recapitulate, $C(n)$ is the sum of the values for square n, $tr(n)$ and $br(n)$ the right hand corners:

$$S = \begin{cases} C(n) = 2(tr(n) + br(n)) \\ tr(n) = (2n-1)^2 \\ br(n) = (2n-1)^2 - 6(n-1) \end{cases}$$

$$\iff C(n) = 16n^2 - 28n + 16.$$

Finally we simply calculate $1 + \sum_{i=2}^{501} C(n)$. The reason why we add a 1 is because the formulate doesn't work $n = 1$ (the single cell), thus we start our sum at 2. Also the sum only goes to 501, not 1001, indeed $501+501-1 = 1001$.

29 Distinct powers

We can use a nifty dodge for this problem. Let's say we have all the terms of the sequence in a list. To get all distinct terms we turn the list into a set,

indeed if you have done a little maths you should know that a set contains unique elements. The Python implementation is simply the *set()* function applied to a list comprehension with two *for* loops.

30 Digit fifth powers

This is one those unbounded problems. What I mean is that the candidate numbers are in $\mathbb{N}^+$. Our first goal is to bound the candidate numbers. Another way of seeing this is to ask what size are the candidates numbers going to be. A number of size n is bounded by $10^n - 1$ (for example a 2-digit number is bounded by 99). Also the sum of the digits of a number of size n to the 5^{th} power is bounded by $9^5 n$. What we are looking is the first integer n which doesn't verify $9^5 n > 10^n - 1$. The solution is simply to try integers by hand, or to use a *while* loop as I did. It turns out that a loose boundary is $6 \times 9^5 = 354294$. Now the problem is simple, we loop through every integer between 2 and 354294 and check its digits to the 5^{th} sum up to the integer with a list comprehension.