Explanation to the first hundred 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. The main objective is to learn, not to be proud of yourself.

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.

14 Longest Collatz sequence

15 Lattice paths

16 Power digit sum

17 Number letter counts

18 Maximum path sum I

19 Counting Sundays

20 Factorial digit sum

21 Amicable numbers

22 Names scores

23 Non-abundant sums

24 Lexicographic permutations

25 1000-digit Fibonacci number

26 Reciprocal cycles

27 Quadratic primes

28 Number spiral diagonals

29 Distinct powers

30 Digit fifth powers

31 Coin sums

32 Pandigital products

33 Digit cancelling fractions

34 Digit factorials

35 Circular primes

36 Double-base palindromes