LINUX VOICE TUTORIAL

EUCLID'S ALGORITHM: RECURSION AND PYTHON

GRAHAM MORRISON

Learn a wonderfully simple algorithm that teaches as much about Python as it does about mathematics.

WHY DO THIS?

- You'll learn ancient wisdom about numbers and their factors.
- This is a great way to see how Python deals with Boolean operations.
- While at the same time, see the danger of putting everything into one line.

We're about to go back to the year 300 BC. A time when much of the world looked like the cover of the Led Zepplin album *Houses of the Holy*. This is the time of Euclid; mathematician, Greek geek and founder of all things geometrical.

The problem that Euclid's algorithm solves is easy enough to understand: what is the largest common divisor of two integers? Take the numbers 100 and 80, for example: what's the largest number that divides into both? You can make some assumptions about what that number might look like – it's going to be even and less that 40, obviously, and maybe more than 20 – but to get any closer is going to require a brute-force approach. Does 25 work? No. 30? Nope. Looks like it might be 20 then, as this divides into both and it doesn't look like there can be a higher number.

How about if the two numbers were 50 and 60? It's not obvious what the common divisor might be for these two, which introduces more guesswork. Or what if the numbers were 123456 and 654321?

Adding and subtracting

For all the non-Euclids, the most basic algorithm may simply halve the smallest number and then start counting down, checking whether the new number divides into both. It will work OK for small values, but it's obviously a computationally expensive approach that will become unrealistic very quickly. There has to be a better way, and that's where Euclid comes in. Euclid discovered that if you compare the smaller number with the difference between the smaller and the larger number, 50 compared to 10 in our second example, and then carried on doing the same comparisons, smaller compared against the remainder of the previous subtraction, until you could continue no further, the previous remainder is the largest common divisor. For the numbers 50 and 60, here's what happens:

> "We're going to use Python, for this, as it's installed on virtually everything"

60 - 50 = 10
50 - 10 = 40
40 - 10 = 30
30 - 10 = 20
20 - 10 = 10
10 - 10 = 0

So the largest common divisor between the numbers 50 and 60 is 10. Try it for yourself. It may get a little longwinded – you could easily see that the solution was going to be 10 in the previous example, for instance, but it always works regardless of how big the numbers are you choose. The next question we should be asking is, why? The solution is to do with common divisors, the group of numbers that can be equally divided into both of our values. The common divisor of **a** (assuming **a** is largest number in the pair), is also a common divisor of **a - b** (assuming **b** is the second number). In the first line of our previous calculation, that's the number 10 (60-50). 10 has its own set of devisors – 1,2, 5 and 10, and this process of subtraction doesn't change the set of common divisors. This makes sense because when you subtract the difference you are subtracting a number that shares the common divisors of both numbers.

It might help if you think about this in terms of reversing the calculations with addition:

10 + 10 = 20

20 shares the common divisors of 10, because we've just doubled it.

20 + 10 = 30

Each addition sharing the same common factor that we started with, until...

50 + 10 = 60

We now have our original two values, and you can see where the common divisors came into the equation and how the reversal of this reveals them.

The next job is to put this idea into code, and you should be able to see that we're on the verge of replacing our numbers with variables anyway, so we just need to add some logic. We're going to use Python for this example, as it's installed on virtually everything – from the Raspberry Pi to Apple's OS X and your Linux distribution. If you've not used the Python interpreter before, just type **python** on the command line and make sure you follow our syntax and indentation exactly. Here's the Python code:

```
def euclid(a, b):
        return b and euclid(b, a%b) or a
```

Woah! Those two lines of code do what we've just spent 700 words trying to explain!

If this is your first foray into Python, we'll try to take it as slowly as we can, starting off with what we've just created. **def euclid(a, b):** defines a function called **euclid** that takes two arguments: **a** and **b**. These values are the same two values we were using before in our explanation. If you've just typed this into Python, you can type **euclid(100,140)** to execute the function.

```
euclid(100,140)
```

The interpreter will spit out the answer, which in this case is 20. Now let's look at what the function is doing, one word or character at a time. **return** is how functions are halted when retuning results from an evaluation. If this line were **return 1234**, the output from the function would always be 1234. But that doesn't include any evaluating, which in our example, is done with the remainder of the line. The next character is **b**, our second number, followed by the word **and**.

Boolean operators

In programming terms, **and** is a Boolean operator. With most other programming languages, for an evaluation to be true both sides of a Boolean **and** need to be non-zero. (1 and 1) is true, for example, whereas (0 and 1) is false, and those languages would typically return a 1 for true and a 0 for false. Python is slightly different in the way it handles return values because it packs more features into a single operation. If the first value is non-zero, it will return the second value from the evaluation. If it's false, it will return the first. Here's a simple function definition and the output from the interpreter to show you what we mean:

```
>>> def andtest(a,b):
...  return a and b
...
>>> andtest(1,2)
2
>>> andltest(0,2)
0
```

This facility gives you the same output you get from other languages – if both values are non-zero, you'll get a non-zero value returned, which is effectively the same as **(1 and 1) = true**. If either the first or the second values are zero, these will be returned, effectively making **(3 and 0) = false**.

But you get more because you get the value of the second number for free, and this is how our code is working. But there's another trick immediately afterwards – recursion:

```
euclid(b, a%b) or a
```

The second argument to the first **and** evaluation calls the function again from within itself. That's the recursion part. The arguments for this second call of the function are the second value itself and the remainder of a division between the first and second number. This remainder of a division, otherwise known as a modulo operation, is a different method to the one we outlined earlier. It's the same theory, only made more efficient. This is because equal divisions of the lower number into the higher number – such as 5 into 28 – help us to fast forward a few steps without losing the common divisor. 28%5=3, which is because 28 divided by 5 = 5, with a remainder of 3. You get the same result as the remainder from the subtractions we were doing earlier, only without all the effort:

28 - 5 = 23
23 - 5 = 18
18 - 5 = 13
13 - 5 = 8
8 - 5 = 3

But when will this recursion stop? When will the function stop calling itself and start returning values back up the chain? That's where the final **or a** comes into play, and it's an evaluation connected to the earlier **and** statement. In most programming languages, an **or** evaluation will only return true if one or the other of the arguments is true – so **(1 or 0)** would equal true, but **(0 or 0)** would be false. In Python, you get better value from the same statement because it returns the first value if it's false and the second value if its not. Here's another quick example from the interpreter:

```
>>> def ortest(a,b):
...  return a or b
...
>>> ortest(1,2)
1
>>> ortest(0,2)
2
```

If the evaluation of the recursively embedded function returns zero, the **and** evaluates the value of **a** against the value of **b**, effectively returning the next to last value for **b** before the final evaluation returned **0**. That's exactly the same result we got when we first worked out Euclid's algorithm manually, but it's quite difficult to imagine. To make things clearer, here's some pseudo code for what happens when we call the function with the values of 60 and 50, showing each recursive step on a line with a number and the values Python is calculating. When a value is finally returned, we change the line number with the returned value inserted into the evaluation so you can see what's happening and how we step back through recursion to the final number:

```
a = 60 b = 50
1: 60 and euclid(50, 10) or 60
2: 50 and euclid(10, 40) or 50
3: 40 and euclid(10, 30) or 40
4: 30 and euclid(10, 20) or 30
5: 20 and euclid(10, 10) or 20
6: 10 and euclid(10, 0) or 10 (RETURNS 10 )
5: 20 and 10 or 20 (RETURNS 10)
4: 30 and 10 or 30 (RETURNS 10)
3: 40 and 10 or 40 (RETURNS 10)
2: 50 and 10 or 50 (RETURNS 10)
1: 60 and 10 or 60 (RETURNS 10)
```

You can test the logic of that comparison yourself without the recursive element:

```
>>> def eval(a,b,c):
...  return a and b or c
...
>>> eval(20,10,20)
10
```

The end result is the product of thousands of years of thought – a concise algorithm that performs a useful operation, all on a single line, while at the same time teaching a little about how Python maximises functionality with its Boolean operations (and also makes itself quite difficult to read in the process).