Algorithms - Series Two(i)

The computer changes epistemology, it changes the meaning of "to understand". To me, you understand something only if you can program it. (You, not someone else!) Otherwise you don't really understand it, you only ***think*** *you understand it.*

Gregory Chaitin Metamaths

N_{th} roots by Newton's Method

Earlier in this series, we looked at Heron of Alexandria's method for finding square roots. Now we are going to use Newton's method to find the n_{th} root of a number.

Newton's method can be used to find the roots of a wide range of equations. In this context, "root" means the value for which the equation equals zero. The following equation will equal zero when y is the cube root of x.

```
f(y) = x - y ** 3
```

More generally, if we are looking for the n_{th} root of x:

```
f(y) = x - y ** n
```

Newton's method works in a similar way to Heron's - we look for the fixed point of an equation. The equation we need is as follows:

```
newton(x) = x - (f(x) / df(x))
```

In this equation, df() is the derivative of f(). In our program we use a "quick and dirty"[1] way to compute this derivative. If dx is some small value, then:

```
df(x) = (f(x + dx) - f(x)) / dx
```

This program is based on some Scheme code from Abelson and Sussman's *The Structure and Interpretation of Computer Programs.* As a result of this, I have made use of lambda expressions. These are very common in Scheme and Python allows us to do pretty much the same thing. Lambda is used to pass functions around our programs. Using lambdas we can write functions which take a function as an argument and return a new function. Here's the start of the code:

```
# root.py - roots, Newton's way.

# some constants used in the program.
DIGITS = 5                  # number of digits to display
DX    = 10 ** -7            # dx is used in calculating the derivative
ERROR = 10 ** -(DIGITS + 1) # margin of error

def main():
    print("*** Roots by Newton's Method. ***""")
    print("Enter a,b to find the bth root of a.")
    print("Enter q to quit.")
    while 1:
        a = input("? ")
        if a == 'q':
            break
        a, b = a.split(',')
        display_root(int(a),int(b))
```

We start by defining some constants that relate to the number of decimal places we want, the margin of error and the value to use for dx. The main() function takes input from the user and then decides whether to quit or send values to display_root().

[1] Readers with a good understanding of calculus might prefer to implement derivative in a more precise way

```
def display_root(a, b):
    """Takes a and b, calls root and displays answer as an integer if possible.
    Otherwise displays it rounded to DIGITS digits."""
    c = root(a, b)
    d = round(c)
    if d ** b == a:
        print(d)
    else:
        k = "%." + str(DIGITS) + 'f'
        print(k % c)
```

This fuction gets a & b from root() and displays the answer as an integer if it can, otherwise it uses a formatted print statement to output the desired number of decimal places.

Now we can get into the nitty-gritty of the code. In the root() function, we see our first lambda. We pass the function that will result in a zero when k is the nth root of x to newton().

```
def root(x, n):
    """Takes integers x & n and returns the nth root of x.
    Passes a function to newton() and a guess of 1."""
    return newton(lambda k: x - k ** y, 1)

def newton(f, guess):
    """Takes a function and a guess.
    Calls fixed_point() on the average-damped function constructed from
    f() and df(x)."""
    df = deriv(f)
    return fixed_point(av_damp(lambda x: (x - (f(x) / df(x)))), guess)
```

Newton() calls deriv() to create a new function: df(). It then sends this to av_damp() to get an average-damped version of the function which is passed along to fixed_point().

```
def deriv(f):
    """Takes a function and returns its derivative."""
    return lambda x: (f(x + DX) - f(x)) / DX

def av(x, y):
    """Returns average of x and y."""
    return (x+y)/2

def av_damp(f):
    """Takes a function and returns average-damped version."""
    return lambda x: av(f(x), x)
```

As you can see, deriv() and av_damp() make use of lambdas, so they are constructing new functions based on the original function passed to them, via newton(), from root(). Fixed_point() and close_enough() are quite straight-forward by comparison:

```
def fixed_point(f, new):
    """Takes a function and a guess.
    Returns the fixed point of the function."""
    old = 0
    while not close_enough(old, new):
        old, new = new, f(new)
    return new

def close_enough(old, new):
    """Returns True if old & new differ by less than ERROR."""
    return abs(old - new) < ERROR

# main
if __name__ == "__main__":
    main()
```

Here's a sample run of the program:

```
>>>
*** Roots by Newton's Method. ***
Enter a,b to find the bth root of a.
Enter q to quit.
? 78125,7
5
? 2,2
1.41421
? q
>>>
```

As you can see, if the result is an integer, we are given it in that form.

If it must be represented as a decimal number, we are given the output to 5 decimal places, as specified by the constants at the start of the program.

There is another reasn why I wanted to show you this method for finding n_{th} roots. In the AKS algorithm for proving primality, which we will look at in a later installment, we need a function which will tell us if a number is a proper power. In other words if it is an integer raised to an integer power. Obviously such numbers are not prime and so can be ruled out on that basis.

Task:

In preparation for the AKS section, can you write a function p_pow(x) which returns True if x is a proper power and False if it is not? The output from an interactive session would look something like this:

```
>>> p_pow(49)
True
>>> p_pow(561)
False
>>> p_pow(16807)
True
```

Alternatively, you could get the function to return False if the number is not a proper power and return a list of the base number and the exponent if it is. This time the output might look like this:

```
>>> p_pow(16807)
(7, 5)
>>>
```

7 raised to the 5_{th} power is 16807.