Algorithms - a simple introduction in Python: Part Four

In my vision, the child programs the computer *and, in doing so, both acquires a sense of mastery over a piece of the most modern and powerful technology and establishes an intimate contact with some of the deepest ideas from science, from mathematics and from the art of intellectual model building.*

Seymour Papert: *Mindstorms*

The first program we are going to use in this section is (hopefully) quite fun. I got the algorithm from a book by Arthur Engel called *Elementary Mathematics from an Algorithmic Standpoint.*

The algorithm in question was created by Christian Zeller, a German mathematician. It tells you the day of the week for a given date.

$$W = D + [2.6M - 0.2] + Y + [Y/4] + [C/4] - 2C$$

Where C is the century, Y the year number within the century, M the month, and D the date. The square brackets indicate that we need only the integer part of the expression. To find the day of the week, you divide W by 7 and take the remainder (this is also known as W modulus 7 or, in Python, W % 7).

There are a couple of peculiarities. By century, we mean the numerical part of the date - so we treat 2012 as being in the 20th century!

Also the numbering of the months is different, March is month 1, so January and February have to be treated as months 11 and 12 of the *previous* year. The days of the week begin on Sunday (ie. Sunday is day 0).

Here's the code.

```
# day_from_date.py
# Uses Christian Zeller's formula as given in Arthur Engel's
# "Elementary Mathematics from an Algorithmic Standpoint."

def main():
    days = ["Sunday", "Monday", "Tuesday", "Wednesday",
            "Thursday", "Friday", "Saturday"]
    print("*** Day of Week ***")
    again = 'y'
    while again !='n':
        date = split_date(get_date())
        while len(date) != 3:
            print("Please try again.")
            date = split_date(get_date())
        print(days[convert(date)])
        again = input("Again? ")

def get_date():
    """Simple input function. Returns a string."""
    date_string = input("Date (dd/mm/yyyy): ")
    return date_string

def split_date(date_string):
    """Takes a string and tries to make it into a list.
        Returns an empty list if the data is entered incorrectly."""
    try:
        date = date_string.split('/')
        for i in range(3):
            date[i] = int(date[i])
        return date
    except (IndexError, ValueError):
        return []
```

```
def convert(date):
    """Takes the date as a list and computes the day of the week."""
    day = date[0]
    (century, year) = divmod(date[2], 100)
    # in the formula, March is month 1
    month = ((date[1] - 3) % 12) + 1
    # January and February are therefore in the previous year!
    if month == 11 or month == 12:
        year -= 1
    a = int(2.6 * month - 0.2)
    b = year // 4
    c = century // 4
    d = 2 * century
    return (day + a + year + b + c - d) % 7

# main part of program
main()
```

Okay, when you have typed that lot in, you need to check that it is working.

Task

I think you should find the day of the week of your birthday and then try these:

- 11/11/1918 (end of WW1).
- 01/01/2001 (first day of this millennium).
- 20/06/1969 (Neil Armstrong walks on the moon).
- 07/06/1954 (Alan Turing died on this day).
- 14/02/2013 (Valentine's Day, 2013!)

You should be able to check these quite easily.

If your program doesn't give the correct answer, you need to do some de-bugging!

Minimum and Maximum

This next algorithm is also taken from Arthur Engel's book. It relies on our old friend "absolute value." This time, though, I am going to use the built-in "abs()" function, for the sake of brevity.

```
# min_max.py
# Demonstrates algorithms for minimum and maximum.

def my_min(x, y):
    """Takes two numbers and returns the smallest."""
    return (x + y - abs(x - y)) / 2

def my_max(x, y):
    """Takes two numbers and returns the largest."""
    return (x + y + abs(x - y)) / 2

# main
if __name__ == "__main__":
    print(" **** Minumum and Maximum ****")
    a = float(input("a: "))
    b = float(input("b: "))
    print("Maximum of a & b: ", my_max(a, b))
    print("Minimum of a & b: ", my_min(a, b))
```

These functions depend upon the formulae you can see printed below (the vertical lines indicate that the absolute value is required).

$$min(x, y) = \frac{x + y - |x - y|}{2}$$

$$max(x, y) = \frac{x + y + |x - y|}{2}$$

Just as in the case of abs(), we don't really need to write these functions, since versions of them exist built-in to Python. I just found the maths of this interesting! In practice, it's a lot easier to use the built-ins (and handy things like "<" and ">"!).

A Better "average()"

Let's try something a little more practical. In the square root program, we defined a function "average(x, y)" which took exactly two arguments and returned the sum of the two numbers divided by two. It would be nice to have a function to which we could pass a variable number of arguments, or alternatively a list, and have it give us the correct result. Here's one way to do this.

```
# average.py accepts variable numbers of arguments or a list.

def average(x, *args):
    """Takes a list or a series of arguments and returns the average."""
    if type(x) == list:
      total = 0
      for n in x:
            total += n
      return total/len(x)
    else:
      for n in args:
            x += n
      return x / (len(args) +1)
```

To have a play with this function in the interpreter, first open a terminal in the folder where the program is saved. Then launch Python and try this:

```
>>> from average import *
>>> average(5)
5.0
>>> average(1, 2, 3, 4, 5)
3.0
>>> a = [4, 5, 6, 7]
>>> average(a)
5.5
```

As you can see, our function is happy to accept a list or a comma-separated series of values. If we pass it a single number, it returns that as the average, which seems reasonable enough!