

# A Bit About Program Design

Think back to the way we wrote the calendar.py program.

At one point we needed to write the following function:

```
def daysSince1800(m,d, y):  
    # This returns the number of days  
    # between 1/1/1800 and m/d/y
```

We did this by calling functions we hadn't written yet:

```
def daysSince1800(m,d, y):  
    # This returns the number of days  
    # between 1/1/1800 and m/d/y  
    total = yearDays(y) + monthDays(m, y) + d-1  
    return total
```

From the calls we know what kinds of parameters these functions will need and what they should return:

```
def yearDays(y):
```

```
    # This returns the sum of the days in  
    # the years from 1800 to y-1
```

```
def monthDays(m, y):
```

```
    # This returns the sum of the days in  
    # all of the months of year y prior to m
```

The process then repeats on each of these functions.

This is called *top-down design*; it is the most common technique for designing programs. It is also a great technique for solving problems in general, even problems that have nothing to do with computers or programs.

Here is a statement of top-down design as a general problem-solving technique:

- Start with the problem at hand
- Decompose the problem into simpler subproblems that together add up to a solution of the original.
- If any of the subproblems are simple enough to be solved directly, give their solutions.
- Repeat this process on all of the remaining subproblems.

The converse of top-down design is *bottom-up design*. Here you start with tools that solve subproblems and look for ways to put them together to solve the main problem. Unless you have a lot of experience this is usually harder to apply to large problems, but it can be handy for details.

For example, in the `calendar.py` program it was inevitable that we would get into a situation where we would need the number of days in each month, so we started with the function

```
def daysInMonth(m, y):  
    # This returns 31 for January,  
    # 28 or 29 for February, etc.
```

If you are stuck on how to write a program, this might give you an easy way to get started.

Programs often write themselves once you get started on them.

## Clicker Question

Here is a design problem you have already faced:

Write function `PatternE(n)`. When `n` is 3 this prints

```
*****  
*  
*  
*  
****  
*  
*  
*  
*****
```

Which of the following is a good breakdown of this into subproblems?

- A) Draw the top half of the E, then the bottom half.
- B) Draw the vertical line of  $2 * n + 3$  stars for the left side of the E, then the horizontal lines for the top, middle and bottom bars.
- C) Draw a line with  $n + 2$  stars for the top, then  $n$  lines with 1 star, then a line with  $n + 1$  stars for the middle bar, then another  $n$  lines with 1 star, then a final line with  $n + 2$  stars for the bottom.
- D) Draw stars that form a figure-8 and erase the ones that aren't part of an E.

For the Mastermind game you need to make a *code* which is a string of 4 random letters taken from "RGBYOP". What is a good way to do this?

- A) Randomly choose a string of 4 letters, then see if they are all in the string "RGBYOP".
- B) Take the first 4 letters from "RGBYOP" and then shuffle them.
- C) Randomly choose an index into "RGBYOP" and put the corresponding letter into an *answer* string; do this 4 times, then return the answer.