Recursive functions:

Functions that are defined in terms of themselves are called recursive functions, or recursively defined functions. This is not as strange as it sounds - a number of well-known functions can be defined in recursive form, and we are all familiar with so-called recursion relations (or difference equations), in which each term in a sequence is defined in terms of previous terms. Some examples:

Exponential function with base \( a \): \( f(n) = a^n \) can be defined by \( f(n) = af(n-1) \), \( f(0) = 1 \)
Factorial function \( f(n) = n! \) can be defined by \( f(n) = n \cdot f(n-1) \), \( f(0) = 1 \).
Fibonacci sequence: \( a_{n+1} = a_n + a_{n-1} \), \( a_0 = 1 \), \( a_1 = 1 \)
(Note that in each case there is an "initial condition" supplied, which is necessary to define the function.)

In computer programming, recursive calculations are the most natural form in which to express functions. These are calculations in which each new calculation is used as input for the next calculation. When we evaluate polynomials by nested multiplication we start with the innermost term and calculate outward, i.e. we calculate \( p(x) = a_0 + a_1 x + \ldots + a_n x^n \) with the recursion \( p_0 = a_0 \), \( p_1 = a_{n-1} + xp_0 \), \( p_2 = a_{n-2} + xp_1 \), \ldots \), \( p_n = a_0 + xp_{n-1} \) is the value of the polynomial - of course when we implement this, we just write \( p = a_k + xp \) because we have no need to retain the intermediate terms of the sequence.

Now in the examples above, we customarily start with the initial condition and work our way out to the value being calculated. This, however, is not explicitly necessary and may not be possible. Instead, the MATLAB programming language, and most other high-level languages, allow functions to be recursively defined, i.e. to use their function values in their own definitions, as long as there is sufficient information for the function values to eventually be resolved. As an example, consider the Fibonacci sequence: We denote the \( n \)th term by \( F(n) \) and write a function for \( F \) in recursive form:

```matlab
function y=F(n)
if n==0|n==1
y=1;
else
y=F(n-1)+F(n-2);
end
```

If we then ask for \( F(10) \), MATLAB executes the function \( F \) and discovers it needs to know \( F(9) \) and \( F(8) \) - then it tries to evaluate \( F(9) \) and \( F(8) \) and discovers it needs to know \( F(7) \) - it follows this chain of function evaluations until it finds \( F(2)=F(1)+F(0)=2 \) and then it works its way back up all the way to \( F(10) \). The number of unresolved function calls that MATLAB will allow at any one time is large (equal to 500, but may be changed by the user) but each such function call uses memory and you can run out of memory (not to mention have a very slow running program) if there are too many. Try calculating \( F(800) \) with the program above and see what happens. So nobody uses this method to actually calculate \( F \), we simply present it as an example.

The more customary and more efficient way to calculate \( F(n) \) would be:

```matlab
function y=F(n)
```
if \( n = 0 \) \( n = 1 \)
  \( y \equiv 1 \);
else
  \( a = 1 ; b = 1 ; \) %.., a, b, are the "last two values", a is the older one
  for \( k = 2 : n \)
    \( y = b + a ; \) %get next term as sum of last two terms
    \( a = b ; b = y ; \) %update last two values
  end

Here are some more recursive versions of functions:
The maximum value in an array is the maximum of the first element and the maximum of all
elements after the first. This gives rise to

function \( y = \text{maxr}(x) \)
  \%find maximum value of array \( x \)
  if length(\( x \)) == 1
    \( y = x \);
  else
    if \( x(1) > \text{maxr}(x(2:end)) \)
      \( y = x(1) ; \)
    else
      \( y = \text{maxr}(x(2:end)); \)
    end
  end

To evaluate \( p(x) = a_0 + a_1 x + \ldots + a_n x^n \) we write it as \( p(x) = a_0 + x[a_1 + a_2 x + \ldots + a_n x^{n-1}] \). This
gives rise to:

function \( y = \text{polyvalr}(p,x) \)
  \%recursive formulation of polynomial evaluation
  \%\( p \) is the array of coefficients
  if length(\( p \)) == 1
    \( y = p(1) \times \text{ones(size}(x)); \)
  else
    \( y = p(1) + x.*\text{polyvalr}(p(2:end),x) \)
  end

The direct way of using this idea is what we did in the program mypolval.m
Note that we do NOT recommend recursive programming in general. There are, however,
certain situations where such a recursive structure is useful. This might be the case where
we're not sure when the calculations will end and there is some stopping condition. You can
always avoid a function calling itself by creating your own "stack" of problems to be solved.