Chapter 16 Functions and procedures

Section 16.1 Review of function terminology

Let's review what we know about function terminology in Maple, as described in Section 6.1.

| Figure 16.1.1 Review of function terminology |
| Terms                                                                 | Example                                                                 |
| 1. Functions process input values and produce an output value. The function definition describes how the values of those parameters are used to compute a function result. When we defined functions using the \( \rightarrow \) (arrow) notation, the result was described as an expression involving arithmetic, structure creators such as \([\ ]\), and built-in (e.g. \(\sin\), \(\sqrt{\text{ }}\), or \(\text{seq}\)) functions or functions previously defined by the user. | \( f := (a, b) \rightarrow \sqrt{a \cdot b} \)  
\( (a, b) \rightarrow \sqrt{a \cdot b} \) \hspace{1cm} (1.1.1) |
| 2. The names given to the symbols used for the input values are called the formal parameters to the function. Functions can have zero or more parameters. There is single output or a result for a function, although we have seen that making that output a list or other compound data structure, we can actually get multiple values as a result. | | |
| 3. Functions are associated with a name by making the function definition the right hand side of an assignment statement. | | |
| 4. After a function is defined, you can get the computer to perform the computation by invoking the function. Invocation involves giving the name of the function, and values for the function's arguments, enclosed in parentheses. The values supplied to the function when it is invoked are called the actual parameters or arguments to the function. | | |
| 5. We can apply a function to each element of a list or other structure via map. This gives us a similar structure of results. | | |

\( f(3.1, \pi) \)  # The actual parameters (arguments)  
# are 3.1 and \( \pi \).  
\( 1.760681686 \sqrt{\pi} \) \hspace{1cm} (1.1.2)
In addition to making it easier to link mathematical purposes with what the computer is doing, programming languages feature functions prominently because of their advantages to the construction of programs. Writing a computation as a function encapsulates it: there is a well-defined starting point given by the values supplied for the parameters, there is a specific computation described within the function definition, and there is a final result to the computation when the function delivers its output. Defining a function does not perform the computation, it just stores the directions for how to do the computation within the computer. Only when the function is later invoked are those actions performed. The function may be invoked multiple times (as would occur when the users invokes it when doing several problems, or when map is used with a function) after it is defined. The expectation of reuse is one of the justifications for the extra work needed to define a function, since re-invocation is typically much easier than even cutting and pasting the actions.

### Section 16.2 Creating a function from a script

script is the term we have used for a reusable computation of several steps. In a typical script, we start with assigning certain variables values. These values are used within the script to compute results typically through one or more statements (assignment, if-then, for/while, etc.). It would be nice to create a function from a script using the -> notation since we already know about it, but due to the limitations of conventional programming language processing, arrow notation can really handle only short one-line functions, not the multi-line scripts we have seen as being typical in solving problems.

In this chapter, we describe how to create a function in Maple where the computational results can be described through a sequence of any number of Maple statements. This will be called creating a Maple procedure. A Maple procedure has the general form

```
function name := proc (formal parameters)
local variables used within procedure;

statements;

return expression or variable containing final output;
end proc;
```

\( \text{map}(f, [3.1, a, 9], 16) \\
[ 7.042726745, 4 \sqrt{a}, 12 ] \) (1.1.3)
Notes
1. There is no semi-colon between the ( ) containing the formal parameters and the word local. If there are no local variables for the procedure, then you may do just proc (formal parameters) ; statements; ....
2. The opposite of a "local variable" is a "global variable". We don't use them much in the Maple procedures we will be working with. You can read more about them if you look up "global" in Maple on-line help. There is an example of the declaration and use of a global variable in Example 16.4.1.

This is where the promise made in Section 4.6 is fulfilled: "we will learn how to write scripts so that no modification of the script is necessary to run it with different versions of the problem". All that will be necessary after creating the procedure will be invoke it repeatedly using different values for the actual parameters (arguments).

To do this, we take the additional steps and conventions to encapsulate the script. Encapsulation of the actions of a script requires identifying and declaring the variables that are inputs by making them parameters to the function, defining the single result of the computation (possibly bundling several computed results into a single list) to be output by the function and identifying the variables that are used only within the script. This latter step is a new one, and allows us to use those variables within the script without worrying about the effect of their use or their assignment will have on those variables within the script.

This is a modification of a script that uses a loop. The original, from chapter 13, computed a sum that approximated $e^x$ and printed out a comparison of the sum with Maple's value for $e^x$. This modification just calculates the final value.

Example 16.2.1 A script to compute a sum

```
#Compute a sum x^i/i! until the term < tol.
s := 1;
tol := 10e-10;
x := .05;
term := 1;
for i from 1 while abs(term)>=tol do
term := term*x/i;
s := s+term;
end do:
s;
```

```
1
1.0 10^{-9}
```
We identify the parameters as \( \textit{tol} \) and \( x \). The local variables are \( s \), \( i \) and \( \text{term} \). The result is in \( s \) after the loop is finished. This leads us to transform the script into a function in this way:

**Example 16.2.2  A procedure to compute a sum**

\[
\text{eApprox} := \text{proc}(x, \text{tol}) \\
\text{local } s, i, \text{term}; \\
\quad s := 1; \\
\quad \text{term} := 1; \\
\quad \text{for } i \text{ from 1 while abs(term)} >= \text{tol do} \\
\qquad \text{term} := \text{term} \times x/i; \\
\qquad s := s + \text{term}; \\
\quad \text{end do}; \\
\quad \text{return } s; \\
\text{end proc;}
\]

\[
\text{proc } (x, \text{tol}) \\
\quad \text{local } s, i, \text{term}, \\
\quad s := 1; \\
\quad \text{term} := 1; \\
\quad \text{for } i \text{ while } \text{tol} <= \text{abs(term)} \text{ do} \quad \text{term} := \text{term} \times x/i; \quad s := s + \text{term} \text{ end do;} \\
\quad \text{return } s \\
\text{end proc}
\]
We can invoke the function in a script that compares the results of the function with Maple's computed value.

### Example 16.2.1 Invoking the procedure after it's been defined

```plaintext
for i from 0 to 2.0 by .1 do
    printf("Computed: %f, Built-in: %f
", eApprox(i, 10^(-5)), \exp(i));
end do;
```

Computed: 1.000000, Built-in: 1.000000
Computed: 1.105171, Built-in: 1.105171
Computed: 1.221403, Built-in: 1.221403
Computed: 1.349859, Built-in: 1.349859
Computed: 1.491824, Built-in: 1.491825
Computed: 1.648721, Built-in: 1.648721
Computed: 1.822118, Built-in: 1.822119
Computed: 2.013753, Built-in: 2.013753
Computed: 2.225541, Built-in: 2.225541
Computed: 2.459603, Built-in: 2.459603
Computed: 2.718282, Built-in: 2.718282
Computed: 3.004165, Built-in: 3.004166
Computed: 3.320117, Built-in: 3.320117
Computed: 3.669296, Built-in: 3.669297
Computed: 4.055199, Built-in: 4.055200
Computed: 4.481689, Built-in: 4.481689
Computed: 4.953032, Built-in: 4.953032
Computed: 5.473946, Built-in: 5.473947
Computed: 7.389055, Built-in: 7.389056
```

We note that there are minor differences between the approximation computed by our function, and that computed by Maple's built-in version of the exponential function because we specified that our loop's tolerance should stop when the terms are $10^{-5}$ in magnitude, even though we are printing six decimal places.

Also note that we are free to use $i$ in the script even though we are also using $i$ in the function. Declaring $i$ to be local within the function means that its use within eApprox is insulated from the
use of that variable outside of the function definition. If we had not declared \( i \) to be local, the function's use of \( i \) would clash with that of script's. This would break the use of \( i \) by the script.

### Section 16.3 Another example of creating a function from a script

Recall this script that printed out numbers in rows and columns, taken from chapter 15:

<table>
<thead>
<tr>
<th>Example 16.3.1 Printing script Example 15.4.1</th>
</tr>
</thead>
</table>

\[
L := \begin{bmatrix} 1.3, 1.4, 9.2 \end{bmatrix}, \begin{bmatrix} 1.9, 2.23, 11.74 \end{bmatrix}, \begin{bmatrix} 1, 2, 3 \end{bmatrix}, \begin{bmatrix} -3.2, 1.7, 10.3 \end{bmatrix}:
\]

numRows := 4:
numCols := 3:

for i from 1 to numRows do
  printf("Row %d:", i);
  for j from 1 to numCols do
    printf("%6.2f", L[i][j]);
  end do;
  printf("\n"); #end the row of output
end do;

Row 1: 1.30 1.40 9.20
Row 2: 1.90 2.23 11.74
Row 3: 1.00 2.00 3.00
Row 4: -3.20 1.70 10.30
We can turn this into a function by identifying the parameters, the local variables, and results:

1. The parameters (inputs) for the script is \( L \), the list of items to be printed, as well as \( \text{numRows} \) and \( \text{numCols} \) describing the layout of the printing.
2. The results: there aren't any, actually. The results are all printing which is a side-effect, rather than something to be returned. To make this function return "nothing" we will have it return \( \text{NULL} \), the null sequence. This is what built-in functions such as \text{print} or \text{printf} that are "pure side-effect" do.
3. The local variables are those that are used only within the script. In this example, this would be \( i \) and \( j \).

Finally, we need to pick a name for the function we are developing. We will call it \( \text{tablePrint} \) as indicative of its purpose.

This means that we will create a function which has the overall form

\[
\text{tablePrint} := \text{proc}(L, \text{numRows}, \text{numCols}) \\
\quad \text{local } i, j; \\
\quad \ldots \\
\quad \text{return } \text{NULL}; \\
\text{end proc}
\]

The details of the function definition are as follows:

Example 16.3.2 Function definition

```
tablePrint := proc(L, numRows, numCols)
    local i, j;
    for i from 1 to numRows do
        printf("Row %d:", i);
        for j from 1 to numCols do
            printf("%6.2f", L[i][j])
        end do;
        printf("\n"); #end the row of output
    end do;
    return NULL;
end proc;
```
proc (L, numRows, numCols)
    local i, j,
    for i to numRows do
        printf("Row %d:", i); for j to numCols do printf("%6.2f", L[i][j]) end do;
        printf("n")
    end do;
    return NULL
end proc

To use this function, we invoke it with values for its parameters. We can invoke it several times, using different values of the parameters. If we forget to include enough parameters, or if their values do not work in the script, we will get an error message.

<table>
<thead>
<tr>
<th>Example 16.3.3 Invocation of function</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Example</strong></td>
<td><strong>Commentary</strong></td>
</tr>
<tr>
<td><code>tablePrint([[1.3, 1.4, 9.2], [1.9, 2.23, 11.74], [1, 2, 3], [-3.2, 1.7, 10.3]], 4, 3)</code></td>
<td></td>
</tr>
<tr>
<td>Row 1: 1.30 1.40 9.20</td>
<td></td>
</tr>
<tr>
<td>Row 2: 1.90 2.23 11.74</td>
<td></td>
</tr>
<tr>
<td>Row 3: 1.00 2.00 3.00</td>
<td></td>
</tr>
<tr>
<td>Row 4: -3.20 1.70 10.30</td>
<td></td>
</tr>
<tr>
<td><code>L := [[1.3, 1.4, 9.2], [1.9, 2.23, 11.74], [1, 2, 3], [-3.2, 1.7, 10.3], [1.3, 2.23, -10.3]]:</code></td>
<td></td>
</tr>
<tr>
<td><code>n := nops(L) :</code></td>
<td></td>
</tr>
<tr>
<td><code>tablePrint(L, n, nops(L[1]))</code></td>
<td></td>
</tr>
<tr>
<td>Row 1: 1.30 1.40 9.20</td>
<td></td>
</tr>
<tr>
<td>Row 2: 1.90 2.23 11.74</td>
<td></td>
</tr>
<tr>
<td>Row 3: 1.00 2.00 3.00</td>
<td></td>
</tr>
<tr>
<td>Row 4: -3.20 1.70 10.30</td>
<td></td>
</tr>
<tr>
<td>Row 5: 1.30 2.23-10.30</td>
<td></td>
</tr>
</tbody>
</table>

In a script executed after the function definition comes into play, we can assign variables and then use them parameters in the function invocation.

Each item in the function evaluation is itself evaluated before handed to the function to compute with. Thus in this case tablePrint's third argument is 3.

Section 16.4 Yet Another example
Recall this problem, done as a script in Chapter 6.

**Figure 16.4.1 A problem from Chapter 6**

(From Sullivan, Pre-calculus, p. 342)

A fast-food restaurant wants a special container to hold coffee. The restaurant wishes the container to quickly cool the coffee from 200°F to 130°F as quickly as possible, and keep the liquid between 110°F and 130°F (optimal drinking temperature) as long as possible. The restaurant has three containers to select from.

(a) The CentiKeeper Company has a container that reduces the temperature of a liquid from 200°F to 100°F in 30 minutes by maintaining a constant temperature of 70°F.

(b) The Temp Control Company has a container that reduces the temperature of a liquid from 200° to 100°F in 25 minutes by maintaining a constant temperature of 60°F.

(c) The Hot'n'Cold Company has a container that reduces the temperature of a liquid from 200° to 120° in 20 minutes by maintaining a constant temperature of 65°F.

How long does it take each container to lower the coffee temperature from 200° to 130°F?

How long will the coffee temperature remain between 110°F and 130°F?

We created the following script to do problem (a)

**Figure 16.4.2 Script to solve problem (a) of Figure 16.1.3.1.**

NewtonEquation := u(t) = T + (u_0 - T) \cdot e^{k \cdot t}

\[ u(t) = T + (u_0 - T) \cdot e^{k \cdot t} \]  
(1.4.1)

constTemp := 70  
70  
(1.4.2)

initPeriod := 30  
30  
(1.4.3)

initTemp := 200  
200  
(1.4.4)

nextTemp := 100  
100  
(1.4.5)

eval(NewtonEquation, \{ T = constTemp, t = initPeriod, u_0 = initTemp \})  
\[ u(30) = 70 + 130 \cdot e^{30 \cdot k} \]  
(1.4.6)

eval((1.4.6), u(initPeriod) = nextTemp)  
100 = 70 + 130 \cdot e^{30 \cdot k}  
(1.4.7)

fsolve((1.4.7), k)  
-0.04887790229  
(1.4.8)
eval (NewtonEquation, \[T=70, u_0 = initTemp, k = (1.4.8)\])

\[u(t) = 70 + 130 e^{-0.04887790229 t}\] \hspace{1cm} (1.4.9)

Time to fall to 130 degrees.
\[\text{fsolve} (\text{rhs}((1.4.9)) = 130)\]

15.81880261 \hspace{1cm} (1.4.10)

Time to fall to 110 degrees:
\[\text{fsolve} (\text{rhs}((1.4.9)) = 110)\]

24.11427130 \hspace{1cm} (1.4.11)

Time to stay between 110 and 130
\[(1.4.11)-(1.4.10)\]

8.29546869 \hspace{1cm} (1.4.12)

\[\text{plot} (\text{rhs}((1.4.9)), t = 0..40, \text{labels} = ["time", "temperature in F"]\)
To turn our "coffee script" into a procedure, we will need to do the same thing as before: identify the variables used as inputs, which variables other than the inputs are used in the script as local variables, and what the results are. We also need to choose a name for the procedure. Most of these findings are highly similar to those of the previous examples and we do not discuss them further. However, there is one difference: there are actually two results computed by the script, the time within the target range of temperatures, and the plot of the temperature graph. Since a function (whose rules of behavior a Maple procedure must follow) returns only one result, we put the two items into a list, and return a list as the result of the procedure.

Example 16.3.4 Coffee script as a Maple procedure
coffeeSolution := proc( constTemp, initPeriod, initTemp, nextTemp)
    local NewtonEquation, k, t, kval,
            initConditionEquation,
            keqn, teqn, texpr, timeInTarget,
            tStart, tEnd, pResult;
    NewtonEquation := u(t) = T+(u[0]-T)*exp(k*t);
    #Determine the value of the constant k in Newton's equation from
    #the data provided by the parameters.
    initConditionEquation := eval(NewtonEquation, [T = constTemp, t = initPeriod, u[0] = initTemp]);
    keqn := eval(initConditionEquation, u(initPeriod) = nextTemp);
    kval := fsolve(keqn, k);
    #Set up equation in t with the computed value of the heat constant.
    teqn := eval(NewtonEquation, [T = constTemp, u[0] = initTemp, k = kval]);
    texpr := rhs(teqn);
    #Compute
    tStart := fsolve(texpr=initTemp,t);
    tEnd := fsolve(texpr=nextTemp,t);
    #Time to stay within initTemp and nextTemp degrees
    timeInTarget := tEnd - tStart;
    pResult := plot(texpr, t = 0 .. 40, labels =
    ["time", "temperature in F"]);
    return ([timeInTarget, pResult]);
end proc;

proc(constTemp, initPeriod, initTemp, nextTemp)
    local NewtonEquation, k, t, kval, initConditionEquation, keqn, teqn, texpr, timeInTarget,
            tStart, tEnd, pResult,
    NewtonEquation := u(t) = T+(u[0]-T) * exp(k*t);
initConditionEquation := eval(NewtonEquation, \[T=\text{constTemp}, t=\text{initPeriod}, u[0] = \text{initTemp}\]);
keqn := eval(initConditionEquation, u(\text{initPeriod}) = \text{nextTemp});
kval := fsolve(keqn, k);
teqn := eval(NewtonEquation, \[T=\text{constTemp}, u[0] = \text{initTemp}, k = kval\]);
texpr := rhs(teqn);
tStart := fsolve(texpr = \text{initTemp}, t);
tEnd := fsolve(texpr = \text{nextTemp}, t);
timeInTarget := tEnd - tStart;
prResult := plot(texpr, t = 0..40, labels = ["time", "temperature in F"]);
return [timeInTarget, prResult]
end proc

#Get the function to do the computation.
resulta := coffeeSolution(70,30,200,100);

#Print out the pieces
printf("Time within target range for (a): %f", resulta[1]);
print(resulta[2]);

[30.00000000, PLOT(...)]
Time within target range for (a): 30.000000
Once we have this function working, we can reuse it to compute solutions to parts (b) and (c) in the problem.

**Figure 16.4.1** Reuse of the Maple procedure to solve parts (b) and (c) of the problem.
Get the function to do the computation. resultb := coffeeSolution(60, 25, 200, 100);

Print out the pieces printf("Time within target range for (b): %f", resultb[1]); print(resultb[2]);

Get the function to do the computation. resultc := coffeeSolution(65, 20, 200, 120);

Print out the pieces printf("Time within target range for (b): %f", resultc[1]); print(resultc[2]);
Time within target range for (b): 20.000000
Section 16.5 Using functions as parameters to other functions

We have observed and made use of the fact that parameters to functions can be any sort of value: integers, floating point numbers, ranges, fractions, expressions. We have also seen a number of situations where functions (i.e. a symbol that has been assigned a function as a value, or which is the name of a built-in function) are meaningful parameters to Maple built-in functions. plot and sort are two built-in functions that can use functions as parameters.

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f := \text{proc} (a) )</td>
<td>( f ) is evaluating the second derivative of temperature in F</td>
</tr>
</tbody>
</table>
local $x$;
return eval(diff((sin(x))^2, x), x = a);
end;

proc(a)
local $x$;
return eval(diff(sin(x)^2, x, x), x = a)
end proc

\[ f(3) = 2 \cos(3)^2 - 2 \sin(3)^2 \]  

plot(f, 1..10)

This is the result of evaluating the second derivative at $x=3$. An alternative form of plot takes a function, rather than an expression. In that case, the second parameter is a range, rather than an equation with a range in it. You can read more about this form in the on-line help within Maple for plot.

\[ \sin^2(x) \] at the value $a$ supplied as its parameter.

Since $f$ is a function, we can use the "old, familiar" form of plot as well and make the first parameter an expression rather than just the name of a function. In that case, we use the "old, familiar" form of the second argument which is an equation with a variable and a range.

We have seen in Table 4.4.8.1 that sort has an optional second argument. If this argument is a function, then that function is used to decide how to order the list to be sorted.
The function to be supplied in this situation should take two arguments. It should return true if the first argument is to be considered greater than the second argument, and false otherwise.

**Example 16.5.2 Using a function in specialized sorting with \texttt{sort}**

\begin{verbatim}
L := [["Keisha", 85], ["David", 46],
     ["Nilesh", 86], ["Jennifer", 81]];
[["Keisha", 85], ["David", 46],
 ["Nilesh", 86], ["Jennifer", 81]]
sortFunc := proc(L1, L2)
        return true;
    else
        return false;
    end if;
end;
proc(L1, L2)
        return true;
    else
        return false;
    end if;
end proc
sort(L, sortFunc)
[["Nilesh", 86], ["Keisha", 85],
 ["Jennifer", 81], ["David", 46]]

sortFunc2 := proc(L1, L2)
    return L1[2] < L2[2];
end;
proc(L1, L2)
    return L1[2] < L2[2];
end proc
sort(L, sortFunc2)
[["David", 46], ["Jennifer", 81],
 ["Keisha", 85], ["Nilesh", 86]]
\end{verbatim}

We have a list of lists. Each item in L is a list consisting of a name and a grade. We wish to sort this list in descending order of grade.

We construct a function that assumes that it is given two items. It returns \texttt{true} if the grade of the first item is better than the grade from the second item, \texttt{false} otherwise.

We invoke the version of \texttt{sort} that takes two arguments. The first is the list to be sorted, and the second the function that determines the order.

We actually could have defined sortFunc in a similar way. The "if-then" notation we used with \texttt{sortFunc} is more long-winded but perhaps helps to more clearly show what is going on. \texttt{sortFunc2} does something similar. It makes use of the fact that Maple will automatically produce \texttt{true} or \texttt{false} when presented with the expression "L1[2]<L2[2]" to evaluate.

This sorts the list in ascending order, rather than descending order.
Section 16.6 Reasons for creating and using Maple procedures and functions

Programmers create procedures because they increase productivity by making it easier to reuse the instructions. As we have seen, reuse is a natural outgrowth of the desires to re-execute instructions on the computer: a) because repeated test execution (with editing) is usually needed to get the program to work correctly in the first place, and b) because engineering design and discovery often leads to wanting to recompute varying parameter values that are interesting or optimal.

The encapsulation has other benefits as well. One is that it makes it easier to look within a worksheet and see the subcomputations if their beginnings and ends are marked by proc, -> or end. Another is that by giving a name to a "piece of computing" it becomes easier to remember and to refer to when talking to others or making plans for future work. Writing a computation as a script invoking a series of functions also provides a "chunking effect" for software development. Each function can be viewed as a separate piece that can be tested individually. This can significantly simplify the overall effort of software development because it's usually simpler to test small pieces and then assemble them, rather than waiting to find problems only after many instructions are entered.

Section 16.7 Some "fine print" details for Maple procedures.

1. By default, Maple's processing of your procedure definition will automatically make variables local that are not parameters. The reason why this happens is because almost all the time the use of variables within a function has nothing to do with their use before or afterwards in the Maple worksheet. Making a variable local (or its use as a parameter) insulates its usage within a function from the usage outside of it. This means that you can reuse a function in any other work without worrying that your usage of variables by the function will wreck what's going on with them outside of the function, and vice versa.

Example 16.4.1 Use of local variables

```
Example 16.4.1 Use of local variables

```

```
In this version of tablePrint2, we forget the declaration of j as a local variable.

```
Warning, `j` is implicitly declared local to procedure `tablePrint2`  

```plaintext
proc (L, numRows, numCols) (1.7.1)
  local i, j;
  for i to numRows do
    printf ("Row %d:", i);
    for j to numCols do
      printf ("%6.2f", L[i][j])
    end do;
    printf ("n")
  end do;
  return NULL
end proc
```

We assign `i` and `j` values before we invoke `tablePrint2`

```plaintext
i := 46;
46 (1.7.2)
```

```plaintext
j := 47;
47 (1.7.3)
```

```plaintext
L := [ [1.3, 1.4, 9.2], [1.9, 2.23, 11.74], [1, 2, 3], [-3.2, 1.7, 10.3], [1.3, 2.23, -10.3]]:
n := nops(L):
```

```plaintext
tablePrint2 (L, n, nops(L[1])) i, j;
46, 47 (1.7.4)
```

As the warning indicates, `j` is nevertheless made local.

This version of `tablePrint` declares `j` to be `global`, so its use within the procedure affects (and is affected) by its external use. As was mentioned in Section 16.2, the use of global variables in procedures is rarely needed and can be very confusing. We use it here to show how confusing it can be.

`tablePrint2` is equivalent to `tablePrint`, so its behavior is the same. Due to encapsulation, the local use of `i` and `j` within the procedure ignores whatever values are given to `i` and `j` outside of the procedure.

When the procedure invocation is over, the external values of `i` and `j` are the same as they were before we invoked `tablePrint2`. The use of `i` and `j` within the procedure, and the values that they took on there, are forgotten.
printf \("n\); \#end the row of output
end do;
return NULL;
end proc;
proc (L, numRows, numCols)
  local i;
  global j;
  for i to numRows do
    printf \("Row \%d:\", i);
    for j to numCols do
      printf \("%6.2f", L[i][j])
    end do;
    printf \("n")
  end do;
  return NULL
end proc
i := 48;
48
(1.7.6)
j := 49;
49
(1.7.7)
L := [ [1.3, 1.4, 9.2], [1.9, 2.23, 11.74], [1, 2, 3], [-3.2, 1.7, 10.3], [1.3, 2.23, -10.3]]:
n := nops(L):

\textit{tablePrint3Bad} (L, n, nops(L[1]))
Row 1: 1.30 1.40 9.20
Row 2: 1.90 2.23 11.74
Row 3: 1.00 2.00 3.00
Row 4: -3.20 1.70 10.30
Row 5: 1.30 2.23-10.30

i, j,
48, 4
(1.7.8)

We have defined \textit{tablePrint3Bad} in a way similar to \textit{tablePrint}, except that \(j\) is a global variable.

We assign \(i\) and \(j\) values before we invoke our procedure.

The procedure works the same way as before, but there is a conspicuous difference: \(j\)'s value has changed afterwards. Contrast this with what happened with \textit{tablePrint2} above.

The value of \(j\) has changed from 49 to what it was set to within the operation of the procedure. It is now 4 because the operation of the inner loop \texttt{for j to numCols do}.

.. made/take on the values 1,2,3 and ultimately 4 which is when the repetition stopped. Since \(i\) was local to \textit{tablePrint3Bad},
its use within that procedure was insulated from its external usage. When the procedure was finished, i's value was restored to what it had been before execution of the procedure. However, the same j is used inside and external to the procedure, so the procedure's use of j had an effect on j after the procedure invocation finished.

2. If there is no return statement, then the result of the procedure will be the last computed result. Procedures may have more than one return statement. Typically this makes sense only when most of the returns occur within an if statement. When a return statement is executed, the function immediately concludes with the output indicated by that return statement. Execution in that case skips over whatever other statements may be within the procedure.

<table>
<thead>
<tr>
<th>Example 16.4.2 A procedure with several returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>This procedure computes the same summation for e^x as before, but has a few extra conditions associated with its operation. If it is able to sum things and find a term that is less than tol in magnitude, then it will do so and return the computed value (as before). However, if the value for x is negative, or if computes 10 terms of the sum and is not able to find a small term, then it will return false.</td>
</tr>
<tr>
<td>The for/while loop will stop if either i goes beyond 10 (to 11), or if abs(term) becomes smaller than tol.</td>
</tr>
<tr>
<td>i will be 11 only if the loop does not stop because of the &quot;while&quot; condition becoming false</td>
</tr>
</tbody>
</table>
sumSeries := proc(x, tol)
# Compute a sum \( x^i/i! \) until the term < tol.
# Procedure returns false if \( x \) is negative or if
# Summation goes on for more than 10 terms without
# producing a term in magnitude less than tol.
    local s, term, i;
    if x<0
        then return false;
    else
        s := 1;
        term := 1;
        for i from 1 to 10
            while abs(term)>=tol do
                term :=
                term*x/i;
                s := s+
                term;
            end do:
            if i=11
                then return false;
            else return s;
        end if;
    end if;
end proc:

sumSeries(-1, 10\(^{-10}\))
false

sumSeries(.1, 10\(^{-6}\))
1.05170917

sumSeries(.1, 10\(^{-30}\))
false

3. The programmer can optionally end a procedure with just **end** instead of **end proc**.
   (Although you can end an if with a "fi" instead of "end if" and you can end a for/while ...do with an "od" instead of of an "end do", **corp** is not allowed as a way of ending the definition of a procedure.)

4. Maple procedures typically do not use parameters to transmit output values. While this is
possible if the parameter is not assigned a value, you will get an error message if you try to do it when the parameter has an assigned value. As a result, we don't do this very much in the examples we give.

Some programming languages (e.g. Fortran, C, or C++) have a way of using parameters both as inputs and as outputs. In those languages, if \( \text{r} \) is assigned the value 5 and then a procedure \( P \) is invoked, it may be the case that \( \text{r}' \)s value after \( P \) finishes will be something other than 5. This can be done in Maple but it is an advanced topic that we won't use yet for our work.

### Example 16.4.2 Trying to use a parameter as a way of outputting a result

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r1 := 'r1'; ) # Make sure ( r1 ) has no value. ( r1 )</td>
<td>We unassign ( r1 ), making sure that it is a pure symbol with no assigned value.</td>
</tr>
</tbody>
</table>
| naughtyProc := proc (x, result) local i, s, term; s := 1; term := x; for i from 1 to 10 do s := s + term*x/i; end do; result := s; return NULL; end proc; naughtyProc(.1, r1); r1; 1.029289682 | We define a procedure that assigns its second parameter the value of the sum.
| naughtyProc(.2, r1); r1; 1.117158730 | We run the procedure, and find that \( r1 \) has the value of the sum afterwards. |
| \( r1 := 'r1'; \) naughtyProc(.1, r1); r1; | We invoke the procedure where the parameter to be assigned already has a value. This time we get an error message, and \( r1 \) is unchanged from what it was before. Confusing? Sure is. Tricks like this are suitable for more experienced programmers who are ready to handle the subtler nuances of evaluation in Maple. |

### Section 16.Z Chapter summary

Figure 16.2.1 General form of a Maple procedure definition
function name := proc (formal parameters)
local variables used within procedure;

statements;

return expression or variable containing final output;
end proc;

Notes
1. There is no semi-colon between the ( ) containing the formal parameters and the word local. If there are no local variables for the procedure, then you may do just proc (formal parameters); statements; ....
2. The opposite of a "local variable" is a "global variable". We don't use them much in the Maple procedures we will be working with. You can read more about them if you look up "global" in Maple on-line help. There is an example of the declaration and use of a global variable in Example 16.4.1.

Example 16.2.2 A procedure to compute a sum

eApprox := proc(x, tol)
local s,i,term;
s := 1;
term := 1;
for i from 1 while abs(term)>=tol do
    term := term*x/i;
s := s+term;
end do;
return s;
end proc;

proc(x, tol)
Example 16.2.1 Invoking the procedure after it's been defined

```plaintext
for i from 0 to 2.0 by .1 do
    printf("Computed: %f, Built-in: %f\n", eApprox(i, 10^(-5)), exp(i));
end do;
```

Computed: 1.000000, Built-in: 1.000000
Computed: 1.105171, Built-in: 1.105171
Computed: 1.221403, Built-in: 1.221403
Computed: 1.349859, Built-in: 1.349859
Computed: 1.491824, Built-in: 1.491825
Computed: 1.648721, Built-in: 1.648721
Computed: 1.822118, Built-in: 1.822119
Computed: 2.013753, Built-in: 2.013753
Computed: 2.225541, Built-in: 2.225541
Computed: 2.459603, Built-in: 2.459603
Computed: 2.718282, Built-in: 2.718282
Computed: 3.004165, Built-in: 3.004166
Computed: 3.320117, Built-in: 3.320117
Computed: 3.669296, Built-in: 3.669297
Computed: 4.055199, Built-in: 4.055200
Computed: 4.481689, Built-in: 4.481689
Computed: 4.953032, Built-in: 4.953032
Computed: 5.473946, Built-in: 5.473947
Computed: 7.389055, Built-in: 7.389056