Maple procedures: writing programs as functions

Chapter Overview

We have already learned *user-defined functions* through use of \( \rightarrow \).

In this chapter, we introduce a way of defining a function allows the use of programming statements, e.g. \( \text{if} \), \( \text{for} \), or \( \text{while} \) through the use of

\[
\text{function name := proc( ...) .... end proc;}
\]

This allows us to develop more complicated functions that use such statements.

Review of function terminology

Let's review what we know function terminology in Maple, as has been developed in the chapters so far.

<table>
<thead>
<tr>
<th>Review of function terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Functions process input values and produce an output value. The function <em>definition</em> describes how the values of those parameters are used to compute a function <em>result</em>. 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{r}}, \text{or seq} )) functions or functions previously defined by the user. The names given to the symbols used for the input values are called the <em>formal parameters</em> 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.</td>
</tr>
</tbody>
</table>
| \( f := (a, b) \rightarrow \sqrt{a \cdot b} \)
\( (a, b) \rightarrow \sqrt{a \cdot b} \) (1.2.1) |
| There are two formal parameters to the function defined above. \( a \), and \( b \) are their names. |
2. Functions are associated by a name by making the function definition the right hand side of an assignment statement.

The function \((a, b) \to \sqrt{a \cdot b}\) is named \(f\) through the assignment.

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.

\[
f(3.1, \pi) \quad \# \text{ The actual parameters (arguments)}
\]
\[
\quad \# \text{ are 3.1 and } \pi.
\]
\[
1.760681686 \sqrt{\pi} \quad (1.2.2)
\]

5. We can apply a function to each element of a list or other structure via \(\text{map}\). This gives us a similar structure of results.

When \(\text{map}\) is given two arguments, the first should be a function, and the second should be a structure. The result is the same kind of structure, but with the function applied to each item in the original structure. \(\text{length}\) is a built-in function that works on, among things, character strings.

\[
\text{map}(\text{length}, ["cylinder", "Mississippi", "sidewinder"])
\]
\[
[8, 11, 10] \quad (1.2.3)
\]

When \(\text{map}\) is given three arguments, the first should be a function, the second a structure, and the third the value that should be the same second argument to each invocation of the function.

\[
\text{map}(f, [3.1, a, 9], 16)
\]
\[
[7.042726745, 4 \sqrt{a}, 12] \quad (1.2.4)
\]

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 user invokes it when doing several problems, or when \(\text{map}\) 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.
Software engineering advantages of functions versus function-less scripts

Almost all programming languages use functions (sometimes called procedures, and, in the case of object-oriented languages such as C++, called methods.) The almost universal prevalence of functions

1. Daisy-chaining functions allows one to build complex behavior a piece at a time. Each piece can be separately tested before linked together with the next piece. This makes testing and debugging simpler because there is less to look at once, as a likely location for error. Simpler development means lower costs for programming more complex tasks.

2. Functions encapsulate an action. There is a clear beginning and end to the statements that provide the action, making it easier to transfer to other scripts or worksheets. The input to the function is explicitly named at the start -- the formal parameters to the function. This makes the function easier to reuse than an ordinary script. Reuse lowers the cost of programming because it allows you to reuse the same programming on multiple jobs. Ease of reuse again means lower costs for programming more complex tasks.

3. Functions give a name to an action, and allow you to invoke the action just by giving its name and the arguments that it should be applied to. This reduces the amount of work if the function is used multiple times in the worksheet. So again, this makes the programming written easier to reuse and therefore brings lower costs for programming more complex tasks.

Creating a short function with proc... end

<table>
<thead>
<tr>
<th>General form of a Maple procedure definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>function name := proc ( formal parameters )</td>
</tr>
<tr>
<td>local variables used within procedure;</td>
</tr>
<tr>
<td>statements;</td>
</tr>
<tr>
<td>return expression or variable containing result of the function;</td>
</tr>
<tr>
<td>end proc;</td>
</tr>
</tbody>
</table>

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 need to put in the semi-colon:
   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 here (Brian: provide hyperlink to reference below using global variable).

A simple function created with -> and with proc

\[
f := (a, b) \rightarrow \frac{\sqrt{a \cdot b}}{2}
\]
\[
(a, b) \rightarrow \frac{1}{2} \sqrt{a \cdot b} \quad (1.4.1)
\]

\[
f(3.0, \pi) \to 0.8660254040 \sqrt{\pi} \quad (1.4.2)
\]

\[
f2 := \text{proc}(a, b); \text{return} \frac{\sqrt{a \cdot b}}{2}; \text{end}
\]
\[
\text{proc}(a, b) \quad (1.4.3)
\]

\[
f2(3.0, \pi) \to 0.8660254040 \sqrt{\pi} \quad (1.4.4)
\]

Creating a function with an if statement in it

An important feature of the proc...end proc way of creating functions is that the code inside the function definition can contain if, while, and for statements.

A short function created by proc end with a control statement

\[
f := \text{proc}(t, a, b, c)
\text{if } t > a
\text{then return } b;
\text{else return } c \cdot t;
\text{end if;}
\text{end proc;}
\text{proc}(t, a, b, c) \quad (1.5.1)
\]

We want to model a situation where a sprinter in a race accelerates at a rate of five meters per second squared for the first two seconds of a race, then travels at a constant velocity of 10 meters/second.
This shows the velocity of the runner at 1 second, 3 seconds, and 1.3 seconds.

This attempt to use the arrow notation to define a function with an "if" fails miserably. You can see that one of the problems is that the computer will get confused because the end of the function definition does not end with the first semi-colon.

Creating a function with local variables

Usually local variables arise when the function needs to compute and store intermediate results that are used for the final answer but are not part of them.

A function with local variables

$LStates1 := [["Pennsylvania", "Harrisburg"],
["Massachusetts", "Boston"], ["California", "Sacramento"],
["Florida", "Tallahassee"], ["Texas", "Austin"]]:$

$LStates2 := [["Berlin", "Berlin"], ["Bavaria", "Munich"],

These are lists whose every item is a list of a state name and a capital city name.

The utility of this function is that we can reuse this kind of max-finding very easily for any number of...
# This function when given a list of pairs of strings, compares
# the length of the second item of each pair and finds the longest.
# It then prints out the corresponding first item.

findLongest := proc (LL)
    local i, maxIndex, maxSoFar;
    maxIndex := 0;
    maxSoFar := 0;
    for i to nops(LL) do
        if maxSoFar < length(LL[i][2])
            then
                maxIndex := i;
                maxSoFar := length(LL[i][2]);
            end if;
        end do;
    return cat("State with longest capital name is: ", LL[maxIndex][1]);
end proc;

findLongest(LStates1);
printf("%s",findLongest(LStates2));

lists of states and capitals. The last line shows how we could get the result to be printed out without the quotation marks.
Designing troubleshooting into your functions

Maple, like many languages has a statement that allows the programmer to stop operation of the program if a situation that shouldn't arise actually happens. The general form of the statement is

```
error, sequence of expressions;
```

When this statement is executed, result of the sequence of expressions (usually some strings describing the nature of the situation) is printed and execution halts. This can be useful to catch mistakes before many more lines of code are executed.

### Another example of a short function created with proc

| fanState := high;  | This short script demonstrates what we want to happen -- if the state of the fan is `high` (the symbol), then we want to assign the fanSpeed to 5000. If the state of the fan is `low`, then, we want fanSpeed to be 2000. |
| speedFast := 5000;  | (1.7.1) |
| speedSlow := 2000;  | (1.7.2) |
| if fanState = high |
| then fanSpeed := speedFast, |
| elif fanState = low |
| then fanSpeed := speedSlow; |
| end if; |
| 5000 | (1.7.3) |
| 5000 | (1.7.4) |
| When we build this into a function, we add a little error-detection into the function itself, so that it will abort operation of whatever code it is running in if it ever encounters a fanState that is neither `high` nor `low`. |
| We have designed a function that gives an error message that is intelligible by design. |
# Returns the value of a fast or slow speed for a fan
# based on the fan state, which should be either the symbol high or low.
setSpeed := proc (fanState)
  local speedFast, speedSlow;
  speedFast := 5000;
  speedSlow := 2000;
  if fanState = high then return speedFast
  elif fanState = low then return speedSlow
  else error "fanState is", fanState, "should be high or low"
  end if
end proc;

setSpeed(high);
setSpeed(low);
setSpeed(middle);
Troubleshooting small function definitions

Function definitions with proc... end proc will exhibit typical "imbalance" error messages such as errors if you forget to balance the typical things that occur in computer syntax: ( and ), if and end if, [ and ], proc and end proc. If you do it in a code edit region, then in addition to printing out an error message, the cursor will pause over the first place where the computer saw the imbalance.

```maple
f := proc (t, a, b, c)
    if a > t then
        return b
    else
        return c*t
    end;
end proc

5000
2000
Error, (in setSpeed) fanState is, middle, should be high or low
```

Seeing this message is an indication that you didn't balance what you started.

We note that there's no "end proc" to end the procedure definition.

After making that correction, we get a message that indicates that the proc wasn't expected. The
f := proc (t, a, b, c)
    if a > t
    then
        return b
    else
        return c*t
    end proc;
end proc;

Error, reserved word `proc` unexpected

f := proc (t, a, b, c)
    if a > t
    then
        return b
    else
        return c*t
    end if;
end proc;

We make that correction, and have no error or warning messages. But we discover that the results are not the same as the previous section.

This is not a balancing problem, but an indication that we are giving incorrect instructions. Looking at the evidence and at the code carefully, we discover that we didn't get the inequality correct.

f(1,2,10,5) 10 (1.8.1)
f(3,2,10,5) 15
f(1.3,2,10,5) (1.8.3)
f := proc (t, a, b, c)
    if a < t then
      return b
    else
      return c*t
    end if;
end proc;

This seems to get it right. It's good to have enough test to thoroughly check out the possibilities.

Creating a function from a script: a general procedure

In a typical script, we start with assigning certain variables values. We have called these variables parameters in that once assigned, we use them through the rest of the script. We can turn a script into a Maple procedure by a process that turns the script parameters into the parameters of the procedure, and the other variables used in the script into the local variables of the procedure. We illustrate this process through the following example.
A script to compute a sum

The original, from chapter 12, 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.

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

1.0 10^-9

0.05

1

1

1.051271096

We identify the parameters as $tol$, and $x$. The local variables are s, i and term. The result is in s after the loop is finished. This leads us to transform the script into a function in this way:

A procedure to compute a sum
The assignment names the function eApprox. We can invoke the function in a script that compares the results of the function with Maple's computed value. By separating the definition of the function from the definition of the printing loop, we make both easier to comprehend. We can tell at a glance that the printing loop prints out the result of eApprox.

| Invoking the procedure after it's been defined |
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

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.

Another example
A problem-solving situation with multiple use of a Maple procedure.

(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)

A Script to solve problem (a) that doesn't define functions or procedures.

\[ \text{NewtonEquation} := u(t) = T + (u_0 - T) \cdot e^{kt} \]
\[ u(t) = T + (u_0 - T) \cdot e^{kt} \]  \(1.10.1\)

\[ \text{constTemp} := 70 \]
\[ 70 \]  \(1.10.2\)

\[ \text{initPeriod} := 30 \]
\[ 30 \]  \(1.10.3\)

\[ \text{initTemp} := 200 \]
\[ 200 \]  \(1.10.4\)

\[ \text{nextTemp} := 100 \]
\[ 100 \]  \(1.10.5\)

\[ \text{eval} \left( \text{NewtonEquation}, [T = \text{constTemp}, t = \text{initPeriod}, u_0 = \text{initTemp}] \right) \]
\[ u(30) = 70 + 130 \cdot e^{30 \cdot k} \]  \(1.10.6\)

\[ \text{eval} \left( (1.10.6), u(\text{initPeriod}) = \text{nextTemp} \right) \]
\[ 100 = 70 + 130 \cdot e^{30 \cdot k} \]  \(1.10.7\)

\[ \text{fsolve} \left( (1.10.7), k \right) \]
\[ -0.04887790229 \]  \(1.10.8\)
\[
eval \left( \text{NewtonEquation, } [T = 70, \ u_0 = \text{initTemp, } k = (1.10.8)] \right)
\]
\[
\ u(t) = 70 + 130 \ e^{-0.04887790229 \ t}
\]  \hspace{1cm} (1.10.9)

Time to fall to 130 degrees.
\[
\text{fsolve} \ (\text{rhs} ((1.10.9))) = 130)
\]
\[
15.81880261
\]  \hspace{1cm} (1.10.10)

Time to fall to 110 degrees:
\[
\text{fsolve} \ (\text{rhs} ((1.10.9))) = 110)
\]
\[
24.11427130
\]  \hspace{1cm} (1.10.11)

Time to stay between 110 and 130
\[
(1.10.11) - (1.10.10)
\]
\[
8.29546869
\]  \hspace{1cm} (1.10.12)

\[
\text{plot} \ (\text{rhs} ((1.10.9)), \ t = 0..40, \ \text{labels} = ["time", \"temperature in F"])
\]
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.

Coffee script as a Maple procedure
coffeeSolution := proc(constTemp, initPeriod, initTemp, nextTemp)
    local NewtonEquation, k, t, kval, initCondEq, 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.
    initCondEq := eval(NewtonEquation, [T = constTemp, t = initPeriod, u[0] = initTemp]);
    keqn := eval(initCondEq, 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:
Once we have this function working, we can reuse it to compute solutions to parts (b) and (c) in the problem.

**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]);

[25.00000000, PLOT(...)]

Time within target range for (b): 25.000000
Time within target range for (b): 20.000000
Summary of the chapter

Brian, this needs to be written.