Chapter 11 Developing simple programs: a series of actions

Section 11.1 Writing a script as a series of actions

We have seen that even some of the easiest technical calculations break down into a series of operations, chained together. Realizing that we will typically need the computer to perform a "series of operations" is the motivation for making the transition to programming, where execution of many operations in a block is typical.

The Maple document interface that we have been using so far easily supports calculations where the user prompts to computer to do a series of steps by positioning the cursor at the first operation and then hitting return (or enter) repeatedly. We now introduce a second way to enter instructions and have them executed as a block. This alternative is often easier to work with when you have a few dozen instructions and will want to execute them all in a chain.

Section 11.1.1 The code edit region

One can open a text field in the Maple document where one can enter a series of instructions. To do this, position your cursor where you want the text field to appear in the document, and with the mouse perform Insert->Code Edit Region
Once the field has been created, you can type the textual version of the Maple instructions you want executed. Each instruction (commonly referred to as a *statement*) must be separated by either a semi-colon, or a colon. If the statement ends in a semi-colon, then its value will be printed during execution of the region. If the statement ends in a colon, then printing of its value will be suppressed just as it is in operation of documents.

Once all the instructions have been entered, you can run them all in a series by typing control-E (command-E on Macintosh), or by entering right-click->Execute Region (on Macintosh, control-click->Execute Region). The results will appear below the region in blue.

**Figure 11.1.1 A code edit region with actions entered, separated by semi-colons.**

<table>
<thead>
<tr>
<th>Code region</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Segments of lines that begin with # are regarded as program comments (for the program reader's eyes), not operations for Maple to carry out.</td>
</tr>
</tbody>
</table>
# Figure 11.1.1
# initialize variables
i := 1;
val := .3; # evaluation point
term := (val^i)/i!; # a term to compute
print("term is", term);
# message
s := term; # s has a copy of the term
tol := 10e-7; # A small value.

1
0.3
0.3
"term is", 0.3
0.3
0.0000010

The results in blue are displayed after we position the cursor in the code edit region and type control-E (command-E on Macintosh), or enter Execute Code Region via the clickable menu.

Result of first assignment (to i)
Result of assignment to val.
Result of assignment to term.
Result of calling the print function.
Result of assignment to s.
Result of assignment to tol.

The code region will develop a scroll bars if the amount of text entered exceeds the size of the window. The size of the field can be adjusted by right click->Component Properties, and then modifying the integers listed for the width and height.

Figure 11.1.2 Enlarging a code Region window by changing component properties via the clickable menu
# Figure 11.1.1
# initialize variables
i := 1;
val := .3;  # evaluation point
term := (val^i)/i;  # a term to compute
print("term is", term);  # message
s := term;  # s has a copy of the term
tol := 10e-7;  # a small value.
The "Collapse Code Edit Region" menu item of the clickable menu will reduce the entire window to an icon. If the first line of the region is a program comment, it will be listed to the right of the icon. Double clicking on the icon will execute the code within.

<table>
<thead>
<tr>
<th>Figure 11.1.3 A collapsed code edit region</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Code region</strong></td>
</tr>
<tr>
<td><img src="image" alt="Code region image" /></td>
</tr>
</tbody>
</table>

**Section 11.1.2 Dealing with the actions in code regions: dealing with typing or execution mistakes.**

Saving your document frequently is a guard against developing a code region but then losing what you typed due to a program crash or other aberrant behavior.

When you execute the code region, previous assignments and definitions you have done in the worksheet still apply unless you explicitly have undone them with a restart; or a new assignment within the code region itself. Thus, if you did \( x := 47 \); in a previous code region window, then \( x \)'s value will still be 47 when you execute a new code region (or execute the same region again) unless you have changed \( x \) since then.
When you open up a previously saved worksheet, you will need to re-execute all code regions (and anything else) which establishes the values of variables you intend to use in your current work. Saving a worksheet saves the text, but wipes out the values stored in variables assigned in the previous session. You will have to Execute->worksheet to get the state of the assigned variables back to what they were when you last used the worksheet.

New kinds of errors and warnings can appear in code regions, due primarily to the new requirement that statements be separated by semi-colons and colons.

For example, if you see

**Warnings** that missing semi-colons are being inserted or
**Error**, missing operator or `;`

you should inspect your code and put in the missing semi-colons. Even if it's only a warning, such omissions will turn into errors when you put additional lines of code into the region.

Code regions also have different behavior when you forget to put in a closing bracket (e.g. a `)` or a `]`). When executing a region you might see

**Warning, premature end of input,** plus some results don't seem to be computed.

or

**Error, `)` unexpected.**

In these situations, you probably provided a closing parentheses instead of an opening parenthesis by mistake, or left out the opening parenthesis. Find it and put it in.

| **Example 11.1.2 Code regions with entry errors** |
|-------------------|--------------------------|
| **Example** | **Commentary** |
| | The problem here is a missing parentheses `)`. |
\[ a := \sin(3.5 + \cos(\pi/2)) \的时代;
\]

Error, `;` unexpected

\[ a := \sin(3.5 + \cos(\pi/2)) \]

Warning, premature end of input, use <Shift> + <Enter> to avoid this message.

This is only a warning, and executing this region does perform the computation.

There's both a missing closing parentheses and a missing semi-colon here.
a := sin(3.5)

Warning, inserted missing
semicolon at end of statement
-0.3507832277

b := cos(3.5)

Error, missing operator or `;`

However, if we add another line without a
semi-colon between, we get just an error.
Both lines are messed up because Maple
needs an explicit separator between the
statements. Just putting the next instruction
on another line is not enough.

Section 11.1.3  Rules of style for writing code, program comments

Everything in the code edit region is just characters. There is no built-in automatic
reformatting. No actions other than what you type at the keyboard or copy/paste with the
mouse will occur other than "Execute Region". There is no way to specify that only part of a
code region is to be done in execution. Output appears outside of the code region into the
worksheet below.
Programs are rarely developed for casual calculation; one develops a program because you expect to execute it several times. Because of this, professional expectations exist for readability and organization in the layout of the program. This allows reuse after a period of time away from the work. It also makes it easier to communicate how to use or extend a program to someone else who wants to use it.

Maple programs are formatted using a simple set of three techniques: with program comments, with blank lines, and with indentation.

A program comment is any part of a line that begins with #. The rest of the line is regarded by Maple as something that people will read, not an instruction to be performed by the computer. Program comments are usually expected to:

1. Give brief descriptions of particular steps or sections, if they are not trivial to grasp to the casual reader.
2. If the program is to be presented formally, give a title and/or author of the program.
   Longer programs might include comments that give an overview and describe how the program can be used.

Blank lines are usually used to indicate the natural conceptual divisions between different groups of instructions. For example, the first few instructions establishing initial values of various might be separated by a few blank lines from the rest of the instructions which establish the main body of work, which in turn may be separated by the "finishing up" instructions.

Indentation is used for indicating nuances of control, such as blocks that are repeated. We will discuss indentation more in the section on while coming up.

Figure 11.1.1 illustrates the use of blank lines and program comments. Example 11.2.1.1 illustrates the use of indentation with a while statement.

Section 11.2 Repetition through "while... do... end do"

Section 11.2.1 A first encounter with while

We need a way of causing many similar actions to be performed, without having to do a lot of typing. We've seen a way of doing this, by using map on a big list of values to serve as input as a function. Here is another way:

Note the following about the example:

1. There is a portion of the program -- the initial part -- where the actions are just done once.
2. The part of the program that is repeated is contained within text that begins while... do and ends end do;
3. The line with the *while* contains a *continuation condition*. Whenever the computer is going to execute the series of statements within the "while... end do", it figures out whether the continuation condition is true. If so, it does the series of statements. If not, it decides that the repetition is finished and moves onto the next instruction after the *end do*.

Like the conditions in piecewise expressions and piecewise functions described in Table 5.4.1 and Example 5.7.1, continuation conditions can contain inequalities or equalities, along with the logical modifiers *and*, *or*, and *not*.

Example 11.2.1 below uses the mathematical fact that for values of $x$ close to 0,

$$
\sin(x) = x - \frac{1}{6} x^3 + \frac{1}{120} x^5 - \frac{1}{5040} x^7 + \frac{1}{362880} x^9 + \ldots
$$

The program computes and adds together the first few terms of this sum, until the next term is less than a specified "small number" $tol$.

<table>
<thead>
<tr>
<th>Example 11.2.1.1</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The initial instructions in the sequence set up the variables $i$, the value of $x$ that we want to use (.3), the value of the first term of the sum, the initial value of the variable that will contain the sum, and the value of the small number. The instructions to be repeated are indented and are found within the <em>while...do... end do</em>. The sequence of four instructions is repeated until the magnitude of term is found to be less than the small number. This check occurs just before the instructions are done for the first time, and then each time after the four instructions are performed. At the very end we print out a message that compares the value of the sum, with Maple's notion of what $\sin(.3)$ is.</td>
</tr>
<tr>
<td>Value of $i$</td>
<td></td>
</tr>
</tbody>
</table>
# Example 11.2.1

# initialize variables
i := 1;
evalpt := .3;
term := (evalpt^i)/i!;
print("term is", term);
s := term; # sum so far
tol := 10e-7;

# add on successive terms
while abs(term) >= tol do
    i := i+2;
    term := (-1)^((i-1)/2) * (evalpt^i)/i!;
    print("term is", term);
    s := s + term;
end do;

print("sum is: ", s, " compared to:", sin(evalpt));

1
0.3
0.3
"term is", 0.3
0.3
0.0000010
3
-0.004500000000
"term is", -0.004500000000
0.2955000000
5
0.00002025000000
"term is", 0.000020250000000
0.2955202500
7
-4.339285714 \times 10^{-8}
"term is", -4.339285714 \times 10^{-8}
Section 11.2.2 Another Example

This program sets up a horizontal line xLine that runs from (0,0) to (100,0) using the line function of the plottools package. It positions a red disk at (0,0). It sets up a table, frame, that will be used to store frames of a movie. The while statement contains instructions that repeatedly invoke a random number generator, which relocates the position of the disk by an amount determined by the random number. It then creates a picture of the disk and xLine, and stores it into the frame. The repetition stops when the horizontal position of the disk goes beyond the value of the variable trackLength (set to 100 in this example).

After the repetition, the contents of the table are converted into a list and a movie is generated.

The reason why a table data structure is necessary to store the frames is because if we used a Vector we would need to know how many frames the movie will have, which we don't, because of the use of random numbers.
# Race to the finish line, 11.2.2

# initialize track and graphics parameters
with(plottools);
with(plots); # load in plotting packages

rad := .5; # parameter for drawing size
trackLength := 100; # length of a line, in units.
maxMove := 6; # maximum size of a random number
frameNumber := 0;

# set up vectors for the x and y position of the spot at the far left of the track.
positionX := 0;
positionY := 0;
frame := table(); # Table to store frames of a movie.
mov := rand(1..maxMove); # Can emit random integers between 1 and maxMove

# Draw a line from (0,0) to (trackLength,0). This will be the track.
xLine := line([0,0],[trackLength,0]);

# Draw the player at the position as a solid disk
player := disk([positionX,positionY], rad, color=red);

# Store the picture in a position in the table.
frameNumber := frameNumber+1;
frame[frameNumber] := display( [xLine,player],axes=none,scaling=constrained);

# Repeatedly, generate a move and draw a new picture.
# Stop repeating when the position moves beyond the end of the track.
while positionX<=trackLength do
    m := move(); # Figure out a move, chosen at random.
    positionX := positionX + m; # Move that player by that amount.

    # Draw the player at the new position as a solid disk
    player := disk([positionX,positionY], rad, color=red);

    # Store the picture in a position in the table.
    frameNumber := frameNumber+1;
    frame[frameNumber] := display( [xLine,player],axes=none,scaling=constrained);
end do;

# Show an animation of all the frames
Section 11.2.3 Yet another example

This example is similar to that of section 11.2.1. It computes successive terms of a sum for the natural logarithm:

\[
\ln(x) = x - 1 - \frac{1}{2} (x - 1)^2 + \frac{1}{3} (x - 1)^3 - \frac{1}{4} (x - 1)^4 + \frac{1}{5} (x - 1)^5 + \ldots
\]

The output is more refined than the initial example, through the use of semi-colons and formatted printing (printf). We might prefer to use printf instead of print results if we were going to show the output of the execution to others, or if the formatting made it much easier for us to comprehend the execution trace.

Many experienced programmers put in informative printing statements into the scripts they develop. They realize that spending a bit of time presenting key information in a more organized, intelligible fashion will save them a lot of time when they are trying to find errors in their programming by looking at execution traces.
# Section 11.2.3
# Computation of logarithms through sums.
# initialize variables
i := 1:  # to be used for successive values of the denominator
val := .3:
xm1 := val-1.0:  # "x minus 1"
xm1p := xm1:  # to be used for powers of x minus 1, initially just xm1
term := xm1:  # next term, initially xm1
s := term;  # variable used to accumulate sum of terms.
tol := 10e-7;
printf("x is %f, tol is %e\n", val, tol);  # print out all values on one line.

# add on successive terms
while abs(term)>=tol do
    i := i+1;  # compute next value of the denominator
    xm1p := -xm1p*xm1;  # Next power of x-1, adjusted by +/- sign.
term := xm1p/i;  # Next term of the sum
    s := s+term;  # Add term to sum.
    printf("term is %e, sum is %f\n", term, s);
end do:
printf("term is %e, sum is %f\n", term, s);
print("sum is: ", s, " compared to: ", ln(val));
print(i, " terms used");

-0.7

0.00000010

x is 0.3000000, tol is 1.000000e-06
term is -2.450000e-01, sum is -0.945000
term is -1.143333e-01, sum is -1.059333
term is -6.002500e-02, sum is -1.119358
term is -3.361400e-02, sum is -1.152972
term is -1.960817e-02, sum is -1.172680
term is -1.176490e-02, sum is -1.184345
term is -7.206001e-03, sum is -1.191551
term is -4.483734e-03, sum is -1.196035
term is -2.824752e-03, sum is -1.198860
term is -1.797570e-03, sum is -1.200657
term is -1.153441e-03, sum is -1.201811
term is -7.453001e-04, sum is -1.202556
term is -4.84451e-04, sum is -1.203041
term is -3.165041e-04, sum is -1.203357
term is -2.077058e-04, sum is -1.203565
term is -1.368415e-04, sum is -1.203702
term is -9.046742e-05, sum is -1.203792
term is -5.999419e-05, sum is -1.203852
term is -3.989613e-05, sum is -1.203892
term is -2.659742e-05, sum is -1.203919
term is -1.777191e-05, sum is -1.203936
term is -1.189946e-05, sum is -1.203948
term is -7.982551e-06, sum is -1.203956
term is -5.364274e-06, sum is -1.203962
term is -3.610569e-06, sum is -1.203965
term is -2.433791e-06, sum is -1.203968
term is -1.642809e-06, sum is -1.203969
term is -1.10312e-06, sum is -1.203970
term is -7.513113e-07, sum is -1.203971

"sum is: ", -1.203971215, "compared to: ", -1.203972804
30, "terms used"


Section 11.3  Developing a script with *whiles*

\subsection*{Section 11.3.1 Program layout style for while statements}

Much time can be saved in comprehending or remembering what a program does by "at a glance" indentation that indicates what is being repeated in a while. Typically the statements that are being repeated are indented more than the enclosing *while*. Example 11.2.1 shows appropriate indentation for a *while*.

\subsection*{Section 11.3.2 Suppressing repetitive printing of values with colons}

The list of all the results being computed in a script is called the *execution trace* or just *trace*. While it is a good idea to see the entire trace initially, at some point the quantity of output induced by the repetitions will work against understanding. If you end a line in a script with a colon, it suppresses output of the result. If you end a while do ... end do with a colon, all the output from the repetitions within will be suppressed, even if none of the statements inside end with a colon.
# Example 11.3.2.1

```plaintext
# initialize variables
i := 1;
evalpt := .3;
term := (evalpt^i)/i!;
print("term is", term);
s := term; # sum so far
tol := 10e-7;

# add on successive terms
while abs(term)>=tol do
    i := i+2;
    term := (-1)^((i-1)/2)*
    (evalpt^i)/i!;
    print("term is", term);
    s := s+term;
end do:

print("sum is: ",s, "compared
to: ", sin(evalpt));
```

## Commentary

The only difference between the program of Example 11.2.1 and this one is that the while statement ends with a colon (after the "end do"). The initial part of the trace is the same, but the part from the repetition contains only the output of the `print` operations inside the repetition.

<table>
<thead>
<tr>
<th>Value of i</th>
<th>Value of evalpt</th>
<th>Value of term (first time)</th>
<th>Side effect of <code>print</code></th>
<th>Value of s (first time)</th>
<th>Value <code>tol</code>, the small number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.3</td>
<td>0.3</td>
<td>&quot;term is&quot;, 0.3</td>
<td>0.3</td>
<td>0.0000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&quot;term is&quot;, -0.004500000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&quot;term is&quot;, 0.00002025000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&quot;term is&quot;, -4.339285714 10^-8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&quot;sum is: &quot;, 0.2955202066, &quot;compared to: &quot;, 0.2955202067</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The message at the end.
Section 11.3.3 Developing scripts with repetitive actions.

When developing a script using the ordinary Document interface, you can use the execution trace to determine whether the sequence of actions is working according to your plans. There are two common problems with this:

1. There is so much output that it becomes hard to find the information you're looking for, or it takes too long for Maple to print out the execution trace.
2. The repetition never stops, even though you thought it would.

For problem (1), there are three suggested tactics:

a) Check first that the portion of the script that occurs before the repetition (before the while) is working. Develop the code in small segments, and test each one as you enter it. For example, if you envision creating a script of 15 lines, you can enter the first three, and execute the script as you have it so far. Verify that it does everything as you intend. Then add the next three lines, and test again. In this way, there's no point in puzzling over a repetition's output if the lead-up hasn't worked. In other words, develop a script incrementally.

b) As you become certain that sections of code work, then, you can replace the semi-colons with colons, to suppress printing of results in that section. In addition, you can put in print or printf statements that will always print out information even if you do a wholesale insertion of colons.

c) If the amount of repetition is controlled by parametric values, then you can adjust those values so that the amount of repetition is reduced during testing. For example, in Example 11.2.1.1, since the amount of repetition depends on the value of tol, you could initially set it to a larger value such as $10^{-3}$ and then set it to a lower value when you are convinced that things are probably running. In code given in Section 11.2.2, you could reduce the output in the initial execution trials by setting trackLength to a smaller number such as 20. Once you are confident that the repetition is working, you can set the parameters back to larger values. In the example, you could set it back to 100.

For problem (2), there are two tactics:

a) Interrupt the execution of the code region by clicking on the "red stop hand" in the Maple toolbar that becomes active when a script is running. This was explained in Section 8.1.

b) Interrupt and inspect the state of execution by entering the Maple debugger. We discuss this latter tactic in the next section.

Section 11.3.4 Finding bugs. An introduction to the Maple debugger

A "bug" is computer terminology for an error in a program. Finding and repairing bugs can take a lot of the programmer's time. Many bugs can be quickly found but some are quite subtle and may take longer to find. Finding mistakes in programs is one of the most difficult skills an introductory programmer learns, and it's mainly through trial and error under guidance from more skilled programmers.

First, have a clear idea of what is correct. Correct behavior by a script does not mean just the absence of error messages or warnings. Just because there are no error messages is not a sufficient sign that all is well -- the values being computed could be the wrong ones. For
example, if you entered \( x := 2*y; \) instead of \( x := 2+y; \) the script would compute the wrong value for \( x \) even though there would be no error messages. The wrong answer would then propagate into further calculations that used the value of \( x \), leading to many more incorrect steps. You need to find the first point where a mistake occurs in execution. Thus you need to know more than just what is right at the end, you need to know what is right at every step.

Next, work so that there are only a few places that are likely spots for the error. In section 11.3.3 we made the suggestion to develop a script incrementally -- entering a few lines, verifying that they work, and then repeating the process until the entire script is working. It is very hard to find errors in a large script typed in entirely before it is executed, because there are so many places where the mistake could be. Experienced programmers build partially, test what they have, and then build some more only if what has happened so far checks out.

If your problem is that execution within a while never seems to finish, you can interrupt the computation with the "red stop hand" (see Section 8.1) and look at the execution trace. Sometimes, inserting extra print or printf statements into the statements being repeated provides enough information for you to diagnose the problem. If this is not sufficient, instead of stopping the computation with the "stop hand", you can interrupt the computation with the Maple debugger, take a few steps to check out something locally, and then quit the computation and fix things before re-execution.

To start the Maple debugger during a long computation, hit the bug icon on the Maple tool bar. A separate window will appear indicating a line from the code region and the current values and variables involved in the computation. The bug icon only becomes active when the script in the code region is actively executing. The bug icon cannot be activated when the code region is not being executed.
Within the debugger window you can enter an arbitrary Maple expression or statement (end with a semi-colon) and the result will be displayed in the window.

Figure 11.3.4.2 Maple debugger window
If you hit the "Next" button of the debugger window, then Maple will do one line of the code region, display the result, and then pause again. You can hit the Next button several times to see a few steps of the computation.

When you are satisfied you have an understanding of what is going on, you can hit the "Quit" button and execution of the code region will end, just as if you had hit the red stop sign.

### Section 11.Z Chapter summary

<table>
<thead>
<tr>
<th>Table 11.Z.1 Code edit regions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Action</strong></td>
</tr>
<tr>
<td>How to create a code region</td>
</tr>
<tr>
<td>How to enter instructions into a code edit region.</td>
</tr>
<tr>
<td>How to execute code you've entered into a</td>
</tr>
<tr>
<td><strong>code region</strong></td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td><strong>How to change the size of a code region</strong></td>
</tr>
<tr>
<td><strong>How to &quot;iconify&quot; a code region</strong></td>
</tr>
<tr>
<td><strong>How to execute code in a code region icon.</strong></td>
</tr>
<tr>
<td><strong>How to &quot;uniconify&quot; a code region icon.</strong></td>
</tr>
</tbody>
</table>

**Table 11.Z.2 Style rules for formatting code in a Code Edit Region**

1. Use blank lines to separate conceptually distinct sections of instructions.
2. Use indentation to visually highlight those instructions that are being repeated in a while loop. See Example 11.Z.1 for an example.
3. Use program comments beginning with # to provide author's notes to the reader of the program. The comments should provide appropriate description of what the program is doing, why, and how. If the program is for wider use, it should also contain information identifying the author just as if it were a paper.

**Table 11.Z.3 Stopping execution**

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Halting a &quot;runaway&quot; or long-running computation</td>
<td>Click on the &quot;red stop hand&quot; icon on the Maple tool bar. You should see a message <strong>Warning, computation interrupted</strong>. This only works if you see <strong>Executing...</strong> in the status window at the bottom left hand corner of the Maple application window, just below all the expression and symbol palettes.</td>
</tr>
<tr>
<td>Entering the Maple debugger</td>
<td>Click on the &quot;bug&quot; icon on the Maple tool bar,</td>
</tr>
</tbody>
</table>
next to the "red stop hand". This only works if you see **Executing...** in the status window at the bottom left hand corner of the Maple application window, just below all the expression and symbol palettes.

Once the debugger window is shown, it will display a line in the middle of the interrupted computation. Clicking on the "Next" button will continue the interrupted computation one more step. You can do this repeatedly to see what is being computed. You can also type in any Maple expression in textual format, and the debugger window will display the value of the expression based on the current values in the computation. Hitting the "Quit" button will stop the computation, and you will see the **Warning, computation interrupted** message in the Maple document.

---

**Table 11.Z.4  Form of a while instruction**

<table>
<thead>
<tr>
<th>some instructions setting things up before the repetition starts</th>
<th>The <strong>condition</strong> can be an inequality such as ( x \leq 17 ), or a combination of things such as ( \text{not(isprime}(i)) ) and ( i &gt; 17 ) or ( (i \mod 47) = 0 ). The <strong>condition</strong> has to be something that Maple can evaluate and decide whether it is true or false. The loop will continue for ever (and have to be interrupted as per Table 11.Z.3) if the <strong>condition</strong> starts off true and never changes as the instructions are repeated. Another way of saying this is the goal in <strong>while</strong> loop writing is to ensure that for some repetition of the instructions, things change enough so that the <strong>condition</strong> becomes false so the repetition will stop on its own, without manual intervention.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>while</strong> condition <strong>do</strong></td>
<td></td>
</tr>
<tr>
<td>some instructions to be repeated while the condition is true</td>
<td></td>
</tr>
<tr>
<td><strong>end do;</strong></td>
<td></td>
</tr>
</tbody>
</table>

---

**Example 11.2.1.1**

<table>
<thead>
<tr>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>The initial instructions in the sequence set up the variables ( i ), the value of ( x ) that we want to use (.3), the value of the first term of the sum, the</td>
</tr>
</tbody>
</table>
# Example 11.2.1

# initialize variables
i := 1;
evalpt := .3;
term := (evalpt^i)/i!;
print("term is", term);
s := term; # sum so far
tol := 10e-7;

# add on successive terms
while abs(term)>=tol do
    i := i+2;
term := (-1)^((i-1)/2)*(evalpt^i)/i!;
    print("term is", term);
s := s+term;
end do;

print("sum is: ",s, "compared to:", sin(evalpt));

Value of \(i\)
Value of evalpt
Value of \(\text{term}\) (first time)
Side effect of \textit{print}
Value of \(s\) (first time)
Value \(\text{tol}\), the small number

Next value of \(i\) (first time repetition)
Next value of \(\text{term}\)
Side effect of \textit{print} statement within \textit{while}.
Next value of \(s\).

Value of \(i\) (second time repetition)
Value of \(\text{term}\) (second time)
Side effect of \textit{print}.
Value of \(s\) (second time)

Value of \(i\) (third time repetition)
Next value of \(\text{term}\) (third time)
\textit{print}
Next value of \(s\) (third time)
0.2955202066
"sum is: ", 0.2955202066, "compared to: ",
0.2955202067

The message at the end.