Scripting and Programming for Modeling, Simulation, and Control

Copyright © Bruce Char, 2008-10
4.8 Troubleshooting scripts ................................................................. 45
4.9 Attachments .................................................................................. 46
   Attachment: Version 2 of sheep script (slightly incorrect), without parameters ........................................ 46
   Attachment: Version 2 of sheep script (corrected), without parameters .................................................... 47
   Attachment: Version 2 of Sheep Script, with parameters ............................................................................. 48
   Attachment: Version 3 of Sheep Script, with parameters ............................................................................. 50
4.10 Summary of Chapter 4 material .................................................. 52
5 Chapter 5 More sophisticated scripting ......................................... 55
5.1 Chapter Overview .......................................................................... 55
5.2 Textual entry of directions compared to palette selection .......... 55
5.3 Lists and plotting. Strings .............................................................. 57
5.4 Learning through on-line documentation and experimentation 60
5.5 Operations on lists ........................................................................ 64
5.6 solve, lists and sequences .............................................................. 65
5.7 What is evaluation? ...................................................................... 66
5.8 Worksheet appearance is not the same as worksheet state ......... 66
5.9 Attachment: Textual versions of terms from the Expression, Common Symbols, and Greek Palettes ..... 66
5.10 Summary of Chapter 5 material ................................................. 67
6 Chapter 6 Using and Defining Functions ......................................... 69
6.1 Chapter overview .......................................................................... 69
6.2 Functions in computer languages: a way of producing an output from inputs ........................................... 69
6.3 How can I remember so many functions? ................................. 70
6.4 Defining your own functions with -> (arrow) ............................. 72
6.5 Troubleshooting function definitions .......................................... 75
6.6 The advantages of using functions in scripts ............................. 76
6.7 Attachment: some built-in problem-solving functions .............. 76
6.8 Summary of Chapter 6 material .................................................. 78
7 Chapter 7 Programming with functions ........................................ 81
7.1 Chapter overview ......................................................................... 81
7.2 Designing functions from context ............................................... 81
7.3 Function composition: daisy-chaining functions together .......... 84
7.4 Expressions with units of measurements: convert .................... 86
7.5 Inputs and outputs to user-defined functions don't have to be numbers ..................................................... 90
7.6 Chapter Summary ........................................................................ 94
8 Chapter 8 Visualization, modeling, and simulation ....................... 97
8.1 Chapter Overview ......................................................................... 97
8.2 Using functions from packages, with ........................................ 97
8.3 plot structures, display and combining plots ............................ 98
8.4 plottools, lines, and other shapes .............................................. 100
8.5 Mathematical models ................................................................. 102
8.6 Drawing x-y position as a function of time through parameterized plots .................................................. 103
8.7 Animations (movies) using animate .......................................... 104
8.8 Controlling animations with the animation toolbar ................. 105
8.9 Another example of animation .................................................... 106
8.10 Exporting animations and non-animated plots .............. 112
8.11 Summary of Chapter 8 ................................................................. 113
9 Moving into programming ............................................................. 115
9.1 Chapter Overview ......................................................................... 115
9.2 More on printing: print, printf and sprintf ............................. 115
9.3 For the curious: why printf is so cryptic ................................. 119
9.4 Code edit regions: executing a series of actions at once .......... 120
9.5 Turning intentions into code ....................................................... 123
9.6 Making code easier to read and understand .............................. 123
9.7 Troubleshooting common errors in entry .................................. 124
# List of Tables

<table>
<thead>
<tr>
<th>Table Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>An early computer</td>
<td>2</td>
</tr>
<tr>
<td>A high-end calculator in 2009</td>
<td>4</td>
</tr>
<tr>
<td>Maple started up with new document in Windows XP</td>
<td>7</td>
</tr>
<tr>
<td>Maple document with first entry area</td>
<td>8</td>
</tr>
<tr>
<td>Math input using 2D Math</td>
<td>8</td>
</tr>
<tr>
<td>Math input with labeled result</td>
<td>8</td>
</tr>
<tr>
<td>Arithmetic operations in Maple</td>
<td>9</td>
</tr>
<tr>
<td>Some examples of Maple error messages</td>
<td>11</td>
</tr>
<tr>
<td>Create Document Block to force a Math input area wherever the cursor is placed</td>
<td>14</td>
</tr>
<tr>
<td>Example of a vocabulary mistake</td>
<td>15</td>
</tr>
<tr>
<td>Maple save menu operation</td>
<td>16</td>
</tr>
<tr>
<td>Maple save dialog box</td>
<td>16</td>
</tr>
<tr>
<td>The Maple state display</td>
<td>17</td>
</tr>
<tr>
<td>Example of plotting</td>
<td>19</td>
</tr>
<tr>
<td>Plot created by right-click -&gt; Plot -&gt; 2DPlot.</td>
<td>19</td>
</tr>
<tr>
<td>User-configured plot using PlotBuilder instead of 2DPlot</td>
<td>20</td>
</tr>
<tr>
<td>The Expression palette. The top of the &quot;Common Symbols&quot; palette is also visible</td>
<td>21</td>
</tr>
<tr>
<td>at the bottom of the picture</td>
<td></td>
</tr>
<tr>
<td>Examples of palette-driven computation</td>
<td>21</td>
</tr>
<tr>
<td>Examples of computing with approximate solving</td>
<td>22</td>
</tr>
<tr>
<td>Evaluate at a point</td>
<td>22</td>
</tr>
<tr>
<td>Operations on equations</td>
<td>23</td>
</tr>
<tr>
<td>Some exact numbers</td>
<td>29</td>
</tr>
<tr>
<td>Exact arithmetic is always exact</td>
<td>30</td>
</tr>
<tr>
<td>Limited-precision (floating point) numbers</td>
<td>30</td>
</tr>
<tr>
<td>Limited-precision (floating point) arithmetic</td>
<td>31</td>
</tr>
<tr>
<td>Two expressions factored</td>
<td>33</td>
</tr>
<tr>
<td>Two equations with &quot;numerical solve&quot;</td>
<td>33</td>
</tr>
<tr>
<td>Contrasting approximate and exact arithmetic</td>
<td>34</td>
</tr>
<tr>
<td>Document after control-T (or Insert-&gt;Text).</td>
<td>37</td>
</tr>
<tr>
<td>Document with a mixture of text and math</td>
<td>38</td>
</tr>
<tr>
<td>control-= puts the results of a calculation in the midst of text</td>
<td>38</td>
</tr>
<tr>
<td>Labeled results</td>
<td>38</td>
</tr>
<tr>
<td>Assignment</td>
<td>39</td>
</tr>
<tr>
<td>= and := mean different things in Maple</td>
<td>40</td>
</tr>
<tr>
<td>Basic data structures in Maple</td>
<td>41</td>
</tr>
<tr>
<td>General format of textual operations in Maple, with examples and mistakes</td>
<td>55</td>
</tr>
<tr>
<td>Plotting of multiple expressions plot(set of expressions, var = range)</td>
<td>57</td>
</tr>
<tr>
<td>Plotting points with lists of numbers</td>
<td>58</td>
</tr>
<tr>
<td>Plot options: colors and labels</td>
<td>59</td>
</tr>
<tr>
<td>Plot command help</td>
<td>61</td>
</tr>
<tr>
<td>Examples of plot</td>
<td>62</td>
</tr>
<tr>
<td>Copying the example</td>
<td>62</td>
</tr>
<tr>
<td>Modifying the example</td>
<td>63</td>
</tr>
<tr>
<td>Plot option help page</td>
<td>63</td>
</tr>
<tr>
<td>Operations on lists</td>
<td>63</td>
</tr>
<tr>
<td>Solving systems of equations with solve</td>
<td>65</td>
</tr>
<tr>
<td>Function results in Maple</td>
<td>69</td>
</tr>
<tr>
<td>Command Completion</td>
<td>70</td>
</tr>
<tr>
<td><strong>A function definition in a</strong> math textbook, and a problem associated with it</td>
<td>72</td>
</tr>
<tr>
<td>(from <em>Precalculus</em>)</td>
<td></td>
</tr>
<tr>
<td><strong>User-defined functions through the Expression Palette</strong></td>
<td>73</td>
</tr>
<tr>
<td><strong>User-defined function textually</strong></td>
<td>74</td>
</tr>
</tbody>
</table>
Error .............................................................................................................................................. 184
To our colleagues and families, who supported us in trail-breaking.

To our students, who learn how to work with the new and different.
1 Introduction -- Technical computing at the turn of the century

1.1 Chapter synopsis

1. We explain what technical computing is about, why it's done with a machine and what kinds of devices are used to do it.

2. The advantages doing calculation with a computer are surveyed.

3. Maple, a system for technical computing is introduced and contrasted with some of the other choices available today.

1.2 What is technical computing? What kinds of technical computing are there?

Well, obviously, it's "computing done for technical work" -- primarily science and engineering, but any other field where mathematical and scientific reasoning is used, which nowadays can include other fields such as finance and business, medicine, or digital media. The success of such reasoning in these fields is well-known. Students of those fields need to become experienced and develop proficiency in how it's done to be able to obtain success in the same way.

You are probably familiar with numerical computing since it's done even without computers, with paper and pencil or with calculators in high school or even earlier. You probably have also seen graphing calculators do simple kinds of graphical computing, also sometimes known as mathematical visualization. In addition to this, computers can and are used to do algebraic computing -- computing where the answer is a formula rather than a number, as well as deductive or logical computing -- using logic or deduction to find answers.

In this course, we will concentrate on numerical, symbolic and graphical computing, although we will see some instances of deductive computing as well.

1.3 What are the advantages of doing technical calculation with a computer?

One advantage is greater quantity of calculation -- computers can do calculations billions of times faster than humans and thousands or millions of times faster than calculator-driven computation. Using them makes some things feasible that are not possible any other way: As you've seen with your high school physics, chemistry, and math courses, scientists and engineers often work with mathematical models -- systems of equations, symbols, and mathematical relations that try to describe key aspects of a situation. Calculation with models helps answer questions about the situation being modeled: How much fuel will be used in this operating scenario? How long will it take to heat up the furnace to 265 degrees? Computer simulation loads a model with some initial conditions, and then through calculation produces predictive results. Simulation can be used to make forecasts such as: how fast will the vehicle be moving after 5 seconds? How far will the pollutants move underground after being buried five years?

Computers make it possible to use models that require much more extensive calculations to reach conclusions. The payoff may be better predictions. Extensive calculation also makes it possible to generate and present more information. Computer visualization goes beyond that to highly detailed pictures or animations of a situation described through a mathematical model. This can lead to better insight and understanding.

Another advantage is that it is easier to make a lasting written record of the work. One of the big differences between professional technical work and homework is that the modeling and calculation has lasting value: it matters to more than one or two people, and it is being used in an on-going project. In a professional situation, a worker may need to go back and review the work a year later, long after the details are faded from easy recall. Other people may want to reuse the programming and will need to be given an explanation about how to use it. Since they have long-term value,
it is beneficial to put them into a form suitable for easy future reference and reuse. Thus the work typically includes both programming and documentary explanation.

1.4 The spectrum of devices for technical computing

Back in the days of the original electronic computers developed during and immediately after World War II, the only kind of device you could use for technical computing (which was the only computing that could be justified in those days) filled up a whole room and needed an extensive staff to support.

An early computer

ENIAC, one of the first electronic computers, being programmed by Herman Goldstine at the University of Pennsylvania circa 1946. (US Army photo) Originally intended to do artillery calculations, it was used for the design and development of the first thermonuclear weapons.

See http://www.seas.upenn.edu/~museum/.

Today's users have a choice of a wide variety of devices:

Personal computers

Typically a computer for individual use can be expected to have the following features:

2. Processing capability of a billion or more arithmetic or memory operations per second. Significant amounts of calculation are used in doing the graphics involved in processing digital media or supporting the artificial worlds presented in computer games. This same power can be used to do the simulation and exploration in many common kinds of scientific and engineering situations. This course is about using that calculation power.

3. Standard selection and pointing devices, keyboard and mice. Some computers support "tablet" operation which allow use of a pen writing on the display. The display area and input devices make it easy to enter information via text or by pointing/selection. While the World wide Web has made much processing possible with just "pointing and clicking", the complexity and mathematical sophistication of the models used in current technical computing outstrip the capabilities of current "point and click" technology. Some (computer programming) language-based interaction is necessary to handle things deftly. This requires a more knowledgeable user, but many people find that the extra power is worth the learning effort.

4. A screen capable of display information equivalent to one or more 8 1/2 x 11 inch pieces of paper. This helps support the development of documentation, or of more complex visualizations.

5. Local storage capable of storing a significant fraction of the text of the books in the Library of Congress. Of course, it's easy for a personal media collection to fill up this kind of storage, but it should not be forgotten that the billions of characters that this storage represents can store years or decades of efforts of computer programmers.
6. Connection to the internet, which makes it easier to communicate and share work with others, and to download new programming. From the viewpoint of technical computing, that it's easy to import and run substantial amounts of programming.

**High performance computers, also known as "supercomputers"**

While hand held devices and personal computers are used widely by the general public for non-technical purposes, there are a class of computers that are used primarily for mathematically-based computing. Typically instead of billions of numerical operations per second ("gigaflops" -- billions of floating point operations), they employ multiple processors in parallel to do trillions of operations ("teraflops") or quadrillions ("petaflops"). The hardware, personnel and energy costs of such computers are significant, so typically they are available through centers used through the Internet from many locations. The kinds of technical problems such devices are used for would include: simulation of complicated physical situations, such as for climate prediction, astrophysics, or engineering design (aircraft, groundwater, large building, automobile).

Using such large devices incur significant expense, from hardware, software and support staff costs, as well as electrical power consumed. For example, the U.S. Department of Energy has an IBM computer at its Los Alamos National Laboratory in New Mexico, USA capable of 1 quadrillion ($10^{15}$) numerical operations per second. It consumes 2345.50 kilowatts when it is running. (reference: http://blog.enterpriseitplanet.com/green/blog/2008/06/green-petaflop-ibms-roadrunner-wins-supercomputer-top-spot.html) The National Center for Computational Sciences (NCCS) at Oak Ridge National Laboratory in Tennessee, USA which has several large supercomputers, has a 2008 annual budget between $80 million and $100 million. (reference: http://news.cnet.com/8301-13772_3-9985500-52.html). A typical personal computer might be capable of approximately 4 billion numerical operations per second, so the Blue Gene computer at Los Alamos provides roughly $25 \times 10^6 = 250,000$ times more computing power.

Typically the programming is developed on personal computers, then moved to the larger devices. Usually the expensive supercomputer time is spent primarily on numerical computation, rather than on providing a nice-to-use interface for users. However, the results of a supercomputer may be shipped over the Internet to a personal computer so that a scientist or engineer may mull over the results in a more contemplative way without incurring additional supercomputer costs.

Multiple computers may be linked together over the Internet to get all the pieces of a particular elaborate computation done. This may have the supercomputers performing the massive numerical computations, while other smaller or personal computers linked in may be displaying the results to a distributed team of investigators who are collectively digesting results and steering the on-going work.

**Hand held or mobile devices**

Calculators are useful for casual computation, where one wants to figure out the solution to a small problem once. It's easy to punch in a few numbers and operations and to read the answer on the display. As inexpensive and small mobile devices, these are typically more limited in:

1. memory (limits to computation size and to built-in features)
2. energy consumption (slower processor speed)
3. form factor (keyboard and display too small for more than casual technical use)

Typically calculators are not networked, making it harder to share results to transfer them elsewhere to continue the work.
A high-end calculator in 2009

The TI-Nspire with CAS is a recent generation calculator from Texas Instruments. It can do numerical and symbolic calculations, as well as graphing. According to ticalc.org (http://www.ticalc.org/basics/calculators/ti-nspire-cas.html) it has 16Mb memory, 20Mb storage and has a 150MHz processor. This makes it have about 100 times less memory, 8000 times less storage, and is about twenty times slower than a typical laptop with a dual core 1.5GHz processor, 2Gb memory, and 160Gb disk. Its screen is 240 x 320 pixels, giving it about twenty times less display area than a typical laptop. This is one of the first generation of calculators with limited wireless networking available as an add-on.


Smartphones, personal digital assistants (PDAs), media players have in theory the same processing capabilities as calculators although most of them are not used extensively for technical computation. One advantage of these kinds of devices is that are typically networked so that it's possible to get a more powerful computer somewhere else in the Internet "cloud" of computational resources to do some of the work.

In the future there may be more convergence of the capabilities of all "mobile small form factor" devices. However the small form factor and the limits to energy consumption will probably continue to constrain the capabilities of such devices compared to larger ones.

**Dedicated controllers**

Even some kinds of toasters have microprocessors in them nowadays. "Smart homes" may network many appliances and home features such as HVAC and lighting control. While these devices are merely configured rather than programmed by end users, students of technical computing should not forget that this is another place where programming is necessary. The economic reasons for switching to computer control of devices appear to be the greater flexibility and variety of control that can be developed at modest cost through programming for many kinds of devices. The kind of programming done for device control often has a mathematical basis. Although the processors in dedicated devices are typically a few orders of magnitude slower than those of personal computers or even smartphones, the programming languages used for them are often the same or similar to those used in personal computing.

**1.5 Maple, a system for technical computing**

In this course, you will learn how to do various kinds of technical tasks using Maple. The first version of Maple was developed in the early 1980s at the University of Waterloo in Ontario, Canada, but has since undergone many refinements and extension. (In Fall 2009, we will be using Maple 13.) Maple's original emphasis was on algebraic computing, so it was used by those who needed formulas as results for their work. In the early '90s it was used extensively in calculus classes as a way of supporting exploratory experimentation and more extensive experience with "applied" problems. More recently it has expanded its domain into the symbolic, numeric, and graphical calculations done more generally in scientific and engineering modeling and simulation.

Today, Maple supports numerical computing and graphical visualization about as well as it does symbolic computing. It has its own programming language but also has ways of doing calculations through the graphical user interface (GUI) that are a kind of augmented "point and click". The default way of interacting with Maple also allows you to mix documentation and computing instructions and results. Thus it's particularly easy in Maple to produce documents that produce a well-documented solution -- a description of the problem, the explanation for how to solve it, and the computations that produce the details of the answer and evidence to justify its correctness.
1.6 What about Systems X, Y, Z, ...?

There are a variety of systems and programming languages used for technical computing nowadays. In a large technical establishment such as a university or research lab, one might find, in addition to Maple the use of systems such as: Python, Matlab, C, Mathematica, , Java, Octave, Macsyma, Sage, Axiom, or Fortran.

The vast array of tasks computers can be used for, the multiplicity of philosophies of system design, and the rapid and unrelenting nature of progress in computing means that there is no "best system" for any situation. Each system has its technical strengths. There may also be historical, cultural, or economic reasons why certain systems are used in certain places which co-exist with the technical justification. As with all popular and demanding activities, there are many products in use.

One important ramification of this is that students in technical fields must expect to become familiar and proficient in more than one programming system. Studying more than one brings greater knowledge of what's possible, and the advantages in judgment that comes from knowledge of diverse ways of achieving a goal. It also allows one to function effectively as a team member in larger cross-organizational efforts and to better to cope with the diversity in computing cultures that's out there.

1.7 Why pick Maple as the first system?

1. It is an interactive system, facilitating quick exploration of new ideas. Compared to languages such as C++ or Java, one can immediately start up an interactive system and get calculation results. It's also possible to enter computational scripts, which are sequences of steps that are less elaborate to set up than programs. The scripts can be easily re-run to do "what if" exploration just by changing a line or two in the script. The worksheet interface to Maple allows sophisticated mathematical typography and graphics as well as mouse- and palette-driven input.

2. It can handle calculations with formulas. Rather than figuring out the formula yourself, you can even get Maple to calculate the formula for you. While many mathematical calculations produce numerical results, formulas are needed to specify how those results are produced. Designers or developers often need to find and create the formulas rather than to just copy them from a book. A system where formulas are easy to represent and easy to create supports this kind of work. A system that allows you to represent and calculate both numbers and formulas makes it easier to do and document technical calculation than a system that just works with numbers.

3. It supports a variety of data structures that support technical computation: formulas, equations, functions, sets, lists, tables, vectors, and matrices. Rather than having calculations done with a number of digits that's fixed forever, it is easy to change the precision if more (or less) precision is needed. Having higher-level interaction with the computer is usually more productive of human time. Having all of these entities as "first class entities" in the system requires less mental effort by the user, since they do not have to translate what they are thinking about (e.g. a table of formulas) into the terms that the computer is using. Rather, the computer language accommodates the human style of thought.

4. It supports documentation as well as calculation. From the instructor's point of view, it's easier to create documents that explain Maple because we can use Maple both to handle the calculation and the presentation of results. For students, having a file with both the directions that cause the computation and the results makes it easier to present the work cleanly for grading, and for future reference. For professionals doing technical work, having an integrated environment where text, programming and results can be presented together can be a convenience.

5. It has a "conventional programming language". An objective of this course is to make you become familiar enough with some of the standard elements of programming (e.g. assignments, conditionals, loops, procedure definitions) so that you can use them creatively to handle certain common situations that can't be handled through point-and-click or built-in operations. These elements will be found in highly similar form in the languages used by many other systems used for technical work.

6. The mathematics of modeling and simulation is an explicit feature of the language. While its programming language is conventional, Maple's language has an important added plus -- you can work over expressions, equations, functions, and other mathematical objects just as conveniently as if they were numbers or characters. Conventional languages (e.g. Java) can represent arithmetic operations -- multiplication, addition, subtraction, and division -- but
after that there are special tricks and conversions that you must perform to bridge the gap between what is written in
the program, and the mathematical ideas that you are trying to use in the model. It's more straightforward in Maple to
represent the mathematical model and the computation based on it. This ease of expression and comprehension by
programmers -- has a hard-headed dollars and sense payoff. Less programmer time ican be spent developing a compu-
tation. There is also an efficiency advantage to explicit representation of formulae: it is possible to use systems such
as Maple to automatically improve the quality and efficiency of simulation calculations. This again leads to lower costs
of doing the work. We think this is a software engineering advantage: it's cheaper in the long run to do technical work
with languages with such features. We believe that most languages supporting technical work will have such function-
ality built-in into them.

1.8 For the curious: using more than one system

Any user of computers who expects to use them professionally for design and investigation must expect to eventually
learn multiple systems. Using computer applications for work is like using tools in a workshop-- you would not expect
to use one tool to do all tasks, even if the tool, like Maple, has "Swiss Army Knife" capabilities. Features that make it
easier to do certain kinds of things may slow down doing other things. Even when systems overlap in features, they
typically have differing philosophies and different technical strengths, which means that certain kinds of work may be
significantly easier in one system than another. For example, developing something in Mathematica or Maple may be
fine and quick for a personal computer, but making the same programming work on a supercomputer may take a lot
of effort. Systems with major development effort behind them (such as Maple and those mentioned in the "section
above) seem to have many similarities and functionality. If major effort were needed to acquire expertise in multiple
technical applications, then prospects would be grim for the computing public. What makes things work out is this: at
the introductory level, the difference between casual computing and professional technical computing is the style of
working (higher emphasis on documentation, justification and ease of reuse), and the use of language-based com-
mands/programming needed to do the more sophisticated operations in technical work. "Crossing over" means getting
over the hurdles of learning the new style of work, and interacting with computers using computer languages. Once
this hurdle is passed, it should take only incremental effort to acquire expertise in the second, third, or n th technical
system. Having formal instruction on the first system should provide an explicit introduction into the concepts and the
work processes to facilitate this.

Most systems realize that they cannot be the sole provider of technical computing services. If Institution A uses
Mathematica and Institution B uses Python for their work, then if they expect to use each other's efforts, there has to
be a way of interconnecting programs written in one system with that of another. Thus most systems have interconnec-
tions. For example, the Matlab Symbolic Toolkit allows Matlab users to call Maple to do formula manipulation. Sim-
ilarly, Maple users can link to Matlab and run a Matlab program they've gotten from a colleague as part of a problem-
solving process written in Maple.

Thus: knowledge of basic programming and the concepts of software development make it possible to switch between
systems with only a modest amount of additional effort. Software interconnection allows one to use efforts done by
others in another system without having to translate. Symbolic computation systems like Maple also have the additional
bonus of being able to translate some of their programming between languages. There is a "convert-to-Matlab" feature
for example, or a "convert-to-C" feature. This conversion doesn't work for computations involving formula manipulation
because C doesn't have that feature. But the conversion does work for other kinds of computations involving just
numbers or text.
2 Getting started with Maple's Document Mode: doing technical work with a clickable interface

2.1 Chapter synopsis

1. How to start up Maple and perform simple calculations, algebra, and plots within it.

2. Detecting and fixing typographical mistakes.

3. Introducing mistakes caused by vocabulary misunderstandings and the use of incorrect logic in giving directions.

2. How to save Maple work so that you can refer to it or resume working on it later.

3. How to recover a Maple worksheet if it or your computer crashes.

2.2 A new document

Start up the Maple application (this varies on the type of computer system you have, typically it involves clicking or double-clicking on the Maple 13 icon, but if you can't figure it out yourself ask for a demo for someone who knows). Once the Maple application window appears, a new "document" will appear in the main working area of the Maple application. A flashing cursor will appear with the outline of a small rectangle with dashed lines. The entry mode will read "Math" and "2D Math". You can make the "quick help" black box disappear by clicking on the "close box" circle-X in the upper right hand corner.

Maple started up with new document in Windows XP

After you close the quick help box, you will see the Maple cursor in a small rectangle with a dashed line outline.
At this point, what you type will appear in the small rectangle and be regarded as a mathematical expression. In the next section, we describe what to type in order to get something useful to happen.

2.3 Exact arithmetic

Grade school arithmetic

In the math area, type 2 + 3. "2 + 3" is regarded as a mathematical expression by Maple when you type it into the dashed rectangle. As you are typing, the input indicators should say that you are entering Math mode, in "2D Math" input:

Math input using 2D Math

This expression should show up in the work area. If you hit the enter key, then Maple will evaluate the expression and you should see the result displayed below the input, as in the figure below:

Math input with labeled result
Maple has automatically calculated the answer and given it a label (1). After Maple calculates the answer, the cursor should appear below the result in another dashed-line rectangle. This indicates that Maple is ready to do another calculation. Maple supports all the basic arithmetic operations in a fashion similar to many other programming languages. One thing that takes some getting used to is that * (asterisk) is used to input multiplication rather than "x" or a centered dot. Another thing that takes some getting used to is that what you type is formatted to look like math notation. Thus if you type a /, Maple understands that you are talking about division and immediately starts formatting your input as if it were a fraction. If you type an asterisk, Maple formats that into a dot (⋅). There is also formatting that occurs with caret (^) since that is the way you enter an exponent in Maple.

### Arithmetic operations in Maple

<table>
<thead>
<tr>
<th>Operation</th>
<th>Character to type/character name</th>
<th>Notes</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>+ (&quot;plus&quot;)</td>
<td></td>
<td>$2 + 2 = 4$ (2.1)</td>
</tr>
<tr>
<td>multiplication</td>
<td>* (&quot;asterisk&quot;)</td>
<td>Typing an asterisk makes a center dot (⋅) appear in the displayed expression.</td>
<td>$2 \cdot 3 = 6$ (2.2)</td>
</tr>
<tr>
<td>division</td>
<td>/ (&quot;slash&quot;)</td>
<td>Typing a slash draws a baseline and then positions the cursor in the denominator. Subsequent typing appears in the denominator. To get out of the denominator and return to normal typing, use the right-arrow key (→). Multiple divisions are by default conducted left-to-right.</td>
<td>$\frac{2}{6} = \frac{1}{3}$ (2.3)</td>
</tr>
<tr>
<td>subtraction</td>
<td>- (&quot;dash&quot; or &quot;hyphen&quot;, typically on the same keyboard key as the underscore. Dash and underscore do not mean the same thing.)</td>
<td>Multiple subtractions are conducted leftmost first.</td>
<td>$3 - 5 = -2$ (2.5)</td>
</tr>
<tr>
<td>parentheses</td>
<td>, (&quot;left parenthesis&quot;, &quot;right parenthesis&quot;)</td>
<td>Use parentheses to change the order of calculation. They are also good for removing any guesswork by the reader as the order of operations.</td>
<td>$(2 + 3) \cdot 5 = 25$ (2.7)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$3 - \left( 5 - \frac{2}{6} \right) = \frac{29}{15}$ (2.8)</td>
</tr>
<tr>
<td>negation</td>
<td>- (&quot;dash&quot; or &quot;hyphen&quot;, typically on the same keyboard key as the underscore). This is the same symbol as used for subtraction</td>
<td>Put a dash in front of a number or parenthesized expression to negate it.</td>
<td>$-(3 \cdot 5 - 2) = -13$ (2.9)</td>
</tr>
<tr>
<td>Operation</td>
<td>Description</td>
<td>Example</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>---------</td>
<td></td>
</tr>
<tr>
<td>caret (^)</td>
<td>Typing a caret moves the cursor to the exponent position. Subsequent typing appears in the exponent. To get out of the exponent and return to the baseline, use the right-arrow key (→).</td>
<td>$2^3 = 8$ (2.10)</td>
<td></td>
</tr>
<tr>
<td>caret (^)</td>
<td></td>
<td>$2^3 - 5$ (2.11)</td>
<td></td>
</tr>
<tr>
<td>caret (^)</td>
<td></td>
<td>$2^{2^{-1}} = \sqrt{2}$ (2.12)</td>
<td></td>
</tr>
<tr>
<td>caret (^)</td>
<td></td>
<td>$2^{-2} + 4 = \frac{21}{5}$ (2.13)</td>
<td></td>
</tr>
</tbody>
</table>

Let's try out some of these operations. We can enter a sum of two fractions by using "+" and "/". If we type 
\[
\frac{2}{3} + \frac{5}{6}
\]
this is what we see:

\[
\frac{3}{2}
\] (2.17)

The way to get a fraction in is to type a slash (/). As soon as you do so, Maple draws an underscore and positions the cursor underneath the fraction line. The next characters you type appear as the denominator. If you type the "+" right after the "3", the plus will appear in the denominator which is permitted by Maple but not what we want in this situation. To get the plus to appear outside of the fraction, we type the right arrow key (the key with → on it). This moves the cursor out of the fraction back into the baseline of the expression. Then we can enter the + for addition, and another fraction. After we hit the enter key, Maple will simplify the result into a single fraction with any common factors removed from the numerator and denominator. Now let's do a multiplication. The Maple programming language (like most) uses an asterisk * as the symbol for multiplication. However, Maple displays the expression with a centered dot. This may be disconcerting -- what you type is not what you see. When you are in "math mode" Maple will be using fancy typography to display whatever math you are entering. We'll see more of this shortly. See if you can reproduce this result:

\[
2 \cdot 3 = 6
\] (2.18)
We can mix operations. Try to enter and calculate the following:

\[
\frac{1 + \frac{2}{3 + 4} + 5 \cdot 6 + 7}{8}
\]

\[
\frac{67}{14}
\]  

(2.19)

In order to get that last denominator, we had to select the expression we had entered for the numerator with the mouse, so that the entire contents of the entry rectangle were blue. Then we typed a slash and the denominator appeared beneath it all. An alternative to using the mouse to enter expression (1.2.1.18) would be to use parentheses. If we type "(1+2/3+4→+5*6+7)/8 enter" we will see this:

\[
\left(1 + \frac{2}{3 + 4} + 5 \cdot 6 + 7\right)
\]

\[
\frac{67}{14}
\]

(2.20)

This allows you to enter complicated expressions without having to use the mouse. Of course, the mouse is still necessary if you want to go back and edit. We observe in passing that a distinctive feature of Maple is that Maple does exact arithmetic with integers and fractions. It keeps fractions as the ratio of two integers. It will, however, automatically simplify such ratios to lowest terms:

\[
\frac{2 \cdot 6 - 18}{3 \cdot 7 - 7}
\]

\[
-2
\]

(2.21)

Making typographical mistakes

Making mistakes is a normal part of using any tool, be it a computer or otherwise. You'll probably make as many mistakes learning how to use Maple as you would make when learning a new sport, a musical instrument, or when learning how to write a good essay. Some of them will be obvious as soon as you make them, others will be subtle or harder to figure out how to fix.

When you make some kinds of mistakes, the computer may give you an error message. For example, if you make a typo and Maple doesn't recognize what you enter as being a valid command, it will complain. Here are some typical error messages. At this stage, some of the messages will make sense and you can react appropriately. Others will use vocabulary that is unfamiliar. For those, the best thing to do is to have a clear idea of what you want to enter (by closely imitating examples that are known to work) and checking carefully that what you have typed was accurate.

Some examples of Maple error messages

```
2 +
Error, invalid sum/difference

2 +
```

We intended to enter "2 + 4" but forgot to type the "4" before we hit enter (return). The appropriate thing to do here is to correct the expression and hit enter again.

```
2 + 4
```
This time we mistakenly enter the expression with the symbols in the wrong order. Maple complains that it expects some operation to be entered between the two numbers but there isn't one.

\[ +2.4 \]

Error, missing operation

\[ +2.4 \]

We intended to enter "2+4" but typed a period instead of a 2 by mistake. Even though we are trying to do the same thing, the error message is different because a different symbol (the period) in this context suggests to Maple that we are trying to do **linear algebra**. The appropriate thing to do here is to correct the expression and hit **enter** again.

\[ 2 + 4 \]

\[ 6 \]

Error, numeric exception: division by zero

If we ask Maple to do an impossible operation, it sometimes gives an error (depending on the operation). The appropriate question to ask yourself here is "what should I be dividing by instead of zero?".

\[ \frac{3}{5 + 3} \]

Error, unable to match delimiters

\[ \frac{3}{5 + 3} \]

We started a sub-expression with a parentheses but forgot to finish it. In Maple, a **delimiter** refers to a parenthesis -- ( or ) -- a bracket [ or ], or a brace { or }. Delimiters are symbols that mark the beginning and end of an expression. In many instances they are necessary to unambiguously indicate meaning. For example \( 5 \cdot (3 + 5) \) evaluates to 40, where as the expression without parentheses \( 5 \cdot 3 + 5 \) means 20 because multiplications are always done before additions unless the parentheses indicate otherwise.

\[ \left( 3 + \left( 5 + \frac{3}{7} \cdot 5 \right) \right) \cdot 2 \]

Error, unable to match delimiters

\[ \left( 3 + \left( 5 + \frac{3}{7} \cdot 5 \right) \right) \cdot 2 \]

This is another instance of the same mistake. We wanted to enter \( \left( 3 + \left( 5 + \frac{3}{7} \cdot 5 \right) \right) \cdot 2 \) but misplaced several parentheses.

\[ 1 + 3 \]

Error, invalid sequence

\[ 1 + 3 \]

We intended to enter "1+3" but typed the extra comma in by mistake. We get an error message that talks about **sequences**, a concept in Maple that we haven't discussed yet (that's coming in the next few chapters). Maple thinks that the sequence is "invalid" because
it usually expects commas to be between items, such as "1,2,x". Maple's language has many other elements in it that we haven't
gotten to discuss yet. If you use any of them by mistake then you will see messages with vocabulary we haven't discussed yet.

It's a fairly typical experience for new users to see some messages that you won't be able to gather much intelligence from other
than the fact that you made a mistake that you should fix,. With knowledge-rich systems such as Maple, you may blunder into
sections of the system that you haven't learned yet. The best strategy is to back out of the situation by editing the expression so
that it is exactly like something that is known to work.

\[
\begin{align*}
2 + \frac{1}{3} \\
\text{Error, invalid fraction} \\
2 + \frac{1}{3} \\
\end{align*}
\]

This one is fairly obvious. In order to fix it though, we need to know what denominator we intended to enter.

\[
\begin{align*}
2 + \frac{9}{3} \\
5 \\
\end{align*}
\]

Correcting typographical mistakes

The standard procedure for fixing a mistake is as you would in a word processor: edit the mistaken input and re-execute
the computation. Here are ways of doing this:

7. Using the mouse, position the cursor where the mistake is. Then use the backspace key to erase the characters
you want to get rid of. Type in more characters to replace it.

8. Use the left arrow key (←) to back up. Typing after backing up then inserts the new typing at the point where the
cursor was positioned.

9. Use the mouse or other "pointing device" of your computer to select a section of what you typed. New typing
then replaces the selection of what you typed.

10. Use the mouse to select a region, then "cut", which you can do through the Maple menu Edit -> Cut. Of course
most people use the keyboard shortcut for cutting, which for Windows or Linux is control-X while on the Mac
it's command-X.

11. Copying and pasting (control/command-C and control/command-V) also works in Maple.

You may find that sometimes you attempt to create a 2D Math input area but Maple does not compute a result for the
input after you hit the enter key. To create a "clickable math" input area if this happens, place the cursor where you
want the input area to be, and use the Format->Create Document Block Menu item of the Maple window:
Create Document Block to force a Math input area wherever the cursor is placed

Exponentiation (powers). Numbers with lots of digits

Use a caret \(^\) to specify an exponent (a "power"). In math mode, Maple will position the cursor so that the next things you enter will become the exponent. As with fractions, you can get out of the exponent by using the right arrow key \(\rightarrow\) when you want to go back to non-exponent numbers. Try entering these expressions

\[
2^3
\]

\[
8
\]

\[
2^{1000} - 2
\]

\[
\begin{array}{c}
5357543035931336604742125245300009052807024058527668037218751941851755255624680612465991894078479290637973364587765734125935726428461570217992288787349287401967283887412115492710537302531185570938977091076523237491790970633699383779582771973038531457285598238843271083830214915826312193418602834034687
\end{array}
\]

We note that Maple does integer and fraction operations exactly. It will not introduce any rounding error into a computation as a calculator would when the answer requires more than ten decimal digits to write down. There are limits to the number of digits Maple will use for integers or fractions, but they have to do more with ultimate limits of the computer hardware and memory rather than a "pre-ordained" decision about how many digits might be useful to keep.

If you type \(\text{kernelopts(maxdigits)}\) into a Math input area on the worksheet, Maple will print out a number which is the maximum number of digits it can handle in any integer or fraction. On the author's computer,

\(\text{kernelopts(maxdigits)} = 268435448\). Note that this is not the value of the maximum number, but how many decimal digits the largest number can have.

For example, Maple can compute the result of
\[
\frac{1}{52!} + \frac{2^{100}}{3^{27}}\] exactly (try it!).

Exact computation is useful not only for doing algebra but also for things such as computing probabilities through counting, or in deriving mathematical formulas for use in simulation or prediction programs.

**Detecting and fixing vocabulary and "logic" mistakes**

There will be other kinds of mistakes where there is no error message, but the response is not what you want. Sometimes this happens because you say something that while grammatical, means something entirely different from what you meant. This could be as simple as entering "2-3" where you meant to type "2+3". Sometimes it is less obvious, such as mistyping the 23-rd digit of a 55 digit number or the 12th term of a long sum. Or worse yet, it could be because you are using the incorrect vocabulary so what you think you are saying does not have that meaning to the computer.

**Example of a vocabulary mistake**

<table>
<thead>
<tr>
<th>2x3 + 5</th>
<th>2x3 + 5</th>
<th>(2.32)</th>
</tr>
</thead>
</table>

Suppose we were under the (mistaken) impression could use "x" in Maple to stand for multiplication. We might use the above as a way to calculate "two times three, plus five". However, while there is no error message, what is calculated is not the number we were expecting. The first step towards proceeding to fix this is to realize that it's not what we want. To Maple, this is a legitimate calculation -- you want to create a formula that's "two times the variable x3, plus five".

| 2·3 + 5 | 11 | (2.33) |

Knowing that the proper way to enter multiplication is through a palette, or symbol "*" (asterisk) as explained in

Finally, there are mistakes made because you ask Maple to do the wrong calculation. Even though Maple does what you want, it turns out that what you wanted was wrong!

For example, you may read a word problem and decide to solve the equation 3·x + 2 = 6, whose solution is x=4/3. But when you plug the solution into the circumstances of the problem, you find that it doesn't solve the word problem. The problem may be because you used faulty logic to decide that 3·x + 2 = 6 was the equation, but it was actually 2·x + 4 = 6. This is known as an "error in logic" or just a "logic error".

Sometimes it's obvious that you made a mistake because the answer is obviously wrong. For example, seeing "-1" when you are expecting the answer to "2+3" is obvious because you know already that you shouldn't get an answer which is a negative number. Sometimes it isn't so obvious, so you need a way to check the correctness of the result.

### 2.4 Saving and retrieving your work

You can save your work in a Maple worksheet file through the File -> Save (keyboard shortcut: control/command-S) menu item of the Maple application. A dialog box appears allowing you to type in the name of the worksheet (we've typed in "test" into the box in the .)

The file is saved with a ".mw" suffix (e.g. test1.mw).

To retrieve the file in a subsequent Maple session, start up the Maple application as before and then use the File->Open menu item to initiate the dialog that retrieves the file.
2.5 Retrieving backups

The "state of Maple" display appears on the bottom left hand side of the Maple window. Usually this display says "Ready", which means that Maple is awaiting your next command. Sometimes it reads "Evaluating..." which indicates that Maple is actively computing an answer. Sometimes it says "Autosaving worksheet", which means that it is saving a copy of the present state of your open worksheets into temporary storage on your computer. The amount of time Maple spends autosaving becomes noticeable in longer Maple sessions when the worksheet contains a lot of results.

Should your computer suddenly lose power or should Maple crash, you can retrieve the last autosaved worksheet by selecting the File -> Recent Documents -> Restore Backup menu item. This will fill your Maple with copies of all the autosaved worksheets. You may then delete them or save them to permanent file space as you wish.
The Maple state display

2.6 Algebra, plotting and mouse-clickable operations

Algebraic expressions and equations. Solving equations. Working with pieces of expressions.

We don’t have to limit ourselves to just numerical calculation in Maple (even if the ability to use many digits and exact fractions allows us to do math more like the math books do). Another distinctive feature of Maple is that you can do algebra by entering expressions with symbols -- the $x, y, z, i$, and $n$ that we see on algebra books. Maple will automatically collect terms and do some simplifications for us automatically

\[ x^2 + 2x + 5 + 3x \]

\[ x^2 + 5x + 5 \] (2.34)

We can even enter equations:

\[ \frac{3}{5}x + 1 = 4 - x \]

\[ \frac{3}{5}x + 1 = 4 - x \] (2.35)

\[ 3x + 1 + 4x = a \cdot x + b \]

\[ 7x + 1 = ax + b \] (2.36)

Note that while Maple automatically collected the $x$ terms on the left hand side of the equation, it does not try to do the more interventionist operation of moving all the $x$ terms to the same side of the equation.

Now, enter the following expression, but rather than hitting the enter key after you’ve entered it, do a right-click. On the Mac, instead of right-clicking hold the control key down then click on the mouse button (this is referred to as "control-click"). A menu of algebraic operations will pop up. Select "factor" and see how Maple can factor the polynomial:

\[ x^2 + 5x - 50 \] \hspace{1cm} \text{factor} \hspace{1cm} (x + 10)(x - 5)

Note that this line does not have a (XX) label for it. To further demonstrate the right-click (control-click) operations available, enter the following equation. Right click on the expression and a pop-up menu should appear that includes a "solve" item. Select the solve and a submenu will appear where you can specify that you want to solve for $x$.

\[ \frac{3}{5}x + 1 = 4 - x \] \hspace{1cm} \text{solve for } x \hspace{1cm} \left[ x = \frac{15}{8} \right]
For the previously experienced: some things are different, for a reason

(This section is for people who already know a programming language and are noticing that Maple does some things differently. Other readers can skip this section.) Most mainstream programming languages work in a similar way. That is why learning Maple has value beyond just being able to use the Maple system -- once you learn Maple, learning Matlab or Java or C is a matter of incremental adjustment. Noticing the differences is educational in that it makes you become aware of the arbitrariness of some of features of languages. If you have used another programming language such as Java or Visual Basic, you will notice that some things do not work the same in Maple as they do there. Some of this can be explained if you know something about the history of programming languages. Some of it is explained by being aware that the designers of Maple had different goals than those who invented Java or VB.

One thing that you have undoubtedly noted is that symbols do not have to have a value associated with them. For example, in Java if you said k=5; Then if you were to create another expression in Java such as System.out.println(k^2 + k + k + 3); then "5" would be used as the value of k in the expression and you would end up printing 38. In Maple, you do not have to associate k with a numerical value before you use k in an algebraic expression. If there is no prior association, Maple just treats the expression as a formula with symbols in it. It may do some algebraic simplification on what you entered, but it does not need to get a number as a result. Since there was no prior assignment for the expressions in section 2.5.1, the calculation done with them just keeps the formula..

Another thing that is different is that in Maple "=" is used for equations, not assignment. The operator in Maple corresponding to "=" in Java or VB is ":=" (a colon immediately followed by an equals, with no spaces inbetween). In Maple, if we wanted to associate "5" with the symbol k, then we would do:

\[ k := 5 \]

\[ 5 \]

\[ k^2 + k + 3 + k \]

38

People who know more than one programming language have a better understanding of which features are change a lot between languages, (such as whether = or := is used for assignment), and which ones are fairly uniform (+ being used for addition, or the use of parentheses in functions and expressions). Maple does not use "=" for assignment because, being a mathematically oriented language that can handle algebra, it wants to make the entry of equations a natural thing. Its use of ":=" for assignment is a feature borrowed from the Algol/Algol 68/Pascal family of programming languages, which picked this operator to make it clear that this operation was not algebraic.

Is ":=" better than ";="? That's a kind of question that is about as hard to answer as trying to decide whether "uno" is better than "one". If one had a language where you had to do ";=*=*=&%#%##%^&==" instead of ";=" or ";="; you could be critical of the choice because it takes much more effort to enter a 35 character operator than a one or two character one. But Maple's choice of ";=" has reasonable motivation -- ";=" is used as a natural way to enter equations (which Java and VB do not support linguistically), and ";=" suggests something similar while being distinctly different. Just as with architectural design of a building, each feature of a programming language is typically carefully considered. Many features are borrowed or copied from predecessor languages, where they have already been subjected to the test of many people using the feature. If you are not familiar with Algol family languages, you can see examples of them at various educational web sites, such as http://www.engin.umd.umich.edu/CIS/course.des/cis400/algol/average.html and http://portal.acm.org/citation.cfm?id=154766.155365).

Plotting and approximate numerical solutions

The right-clickable interface can also activate a plot of a formula you may have entered. Enter this formula, then right-click and select Plot ->2d plot. The automatic defaults for plotting this produce this result.
Example of plotting

If we click on the plot and then position the mouse over the plot area, we see in the upper left hand corner of the Maple application a pair of coordinates that change as we move the mouse around. We can "eyeball" the plot with this to find approximately where this formula is equal to zero. From the figure below we can see that \( x^2 - 10x + 4 \) is zero at about .5 and 9. We could position the cursor in those areas to get a more precise (but still rough) estimate.

Plot created by right-click -> Plot -> 2DPlot.

User has clicked on the plot and positioned the cursor at the coordinate (-3.08, 45.78). The cursor was not captured by the screenshot although it is visible under ordinary use.
The 2DPlot operation makes pre-set decisions about the plot, such as the range of $x$ (-10 to 10), the color of the line, axes labelling, etc. Users can inject their own preferences about these things about this by selecting right-click -> Plot -> PlotBuilder and filling in the dialog box with their choices.

**User-configured plot using PlotBuilder instead of 2DPlot**

The Expressions Palette and the Common Symbols Palette: entering Trigs, logs, roots, exponentials

It's possible to get the common functions of high school algebra, pre-calculus, and calculus by using the Expression palette of the Maple Window:
The Expression palette. The top of the "Common Symbols" palette is also visible at the bottom of the picture

For example, to enter the square root of 36, click on the palette entry for $\sqrt{a}$. That expression will appear in the document, with the "a" selected. If you then type 36 on the keyboard, that number will replace the selected text. If you then hit the enter key, Maple will evaluate the expression and produce the exact result "6".

$$x + y + \frac{1}{2} + \frac{1}{4} + \sqrt{36}$$

$$x + y + \frac{27}{4}$$

You can use the palette multiple times, to create more complicated expressions. Just continue to the use mouse or the arrow keys to move around in the expression. Selecting and typing, backspacing or deleting are all ways of replacing or correcting pieces of the expression. The Common symbols palette, two panels below the Expression palette, can be used to enter $\pi$ and $e$, the base of the natural logarithm system. The palette does not have the inverse trig functions, so you have to enter them through typing. Their names are: arcsin, arccos, arctan, etc.

Examples of palette-driven computation

$$\sin\left(\frac{\pi}{4}\right) + \cos\left(\frac{\pi}{4}\right)$$

$$\sqrt{2}$$

$$\left(\sqrt{1024} + \ln\left(\frac{3}{13}\right)\right) \cdot \pi$$

$$\frac{98}{39} \pi$$
Approximate numerical (calculator-type) arithmetic in Maple

If you enter expressions with integers, exact fractions, and symbols such as $\pi$ and $e$, then Maple will perform exact calculations rather than give approximate answers as a conventional calculator would. You can get approximations by selecting the "numerically solve" instead of the "solve" option from the right-click pop-up menu.

**Examples of computing with approximate solving**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Solve</th>
<th>Approximate Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x^2 - 10x + 4 = x^2 - 10x + 4$</td>
<td>solve</td>
<td>( x = 5 + \sqrt{21}, x = 5 - \sqrt{21} )</td>
</tr>
<tr>
<td>$x^2 - 10x + 4$</td>
<td>solve</td>
<td>0.4174243050, 9.582575695</td>
</tr>
</tbody>
</table>

If you have an exact expression, you can ask Maple to approximate it to 5, 10, 20, or more digits. In this mode, Maple can be used as a super-accurate calculator.

**Examples of numerical computation**

1. Enter fraction, select approximate->20 from right-click pop-up menu.
   \[
   \frac{47}{52} + \frac{4}{3} \quad \text{at 20 digits} \quad 2.2371794871794871795
   \]

2. Enter exact expression, select approximate->5 from right click pop-up menu.
   \[
   \sin\left(\frac{\pi}{10}\right) \quad \text{at 5 digits} \quad 0.30902
   \]

3. Enter equation. Then solve->solve, then select Element->1, then right hand side, then approximate->10
   \[
   \sin\left(\sqrt{e}\right) = \frac{1}{3} \quad \text{solve} \quad \left\{ x = 2\ln\left(\arcsin\left(\frac{1}{3}\right)\right) \right\} \quad \text{2.158578910}
   \]

**Evaluation at a point, and selection of pieces**

Sometimes you wish to evaluate an expression for a particular value of a variable. There is a right-click operation that does this.

**Evaluate at a point**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Evaluate at Point</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x^2 - 2a\cdot x = 0 )</td>
<td>evaluate at point</td>
<td>( x = \frac{1}{4} - a = 0 )</td>
</tr>
<tr>
<td>( x^2 - 2a\cdot x = 0 )</td>
<td>evaluate at point</td>
<td>( x^2 - 6y^2\cdot x = 0 )</td>
</tr>
<tr>
<td>( 3y + 5 )</td>
<td>evaluate at point</td>
<td>14</td>
</tr>
</tbody>
</table>

This operation will give a pop-up menu that will allow us to choose values for all the variables. In the first example, we picked 1/2 for a value of x. Note that the pop-up menu will show what you typed rather than displaying 2D math. In the second example, we specified "3*y=2" as the value for a. In the third example, we picked 3 as the value for y.

Expressions often have pieces. Using the right-click menu, it's possible to select or extract a portion of an expression for further work.
Operations on equations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>right hand side, left hand side</td>
<td>$x = \frac{\sin(a)}{r^2 - 1}$</td>
</tr>
<tr>
<td>right hand side</td>
<td>$\frac{\sin(a)}{r^2 - 1} = x$</td>
</tr>
</tbody>
</table>

One of the options in the right-click menu is "right hand side". It only works for equations.

Operations on multi-part expressions

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x^2 - 4 \cdot x = 4$</td>
<td>solve for x: $[x = 2 + 2\sqrt{2}, [x = 2 - 2\sqrt{2}]]$</td>
</tr>
<tr>
<td>$x = 2 + 2\sqrt{2}$</td>
<td>select entry 1 at 5 digits $x = 2 + 2\sqrt{2}$</td>
</tr>
<tr>
<td>$x = 4.8284$</td>
<td>select entry 1 at 5 digits $x = 2 + 2\sqrt{2}$</td>
</tr>
</tbody>
</table>

Solving this quadratic equation reveals that there are two solutions. Right clicking on these selections and then select entry $\rightarrow 1$ produces the first solution. We can then approximate it by using the

# 2.7 A quick-reference summary to this chapter

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2 + \frac{3^2}{4} - \frac{1}{6}$</td>
<td>Use +, *, -, /, ^ for arithmetic. Hitting the Enter key produces a labelled result.</td>
<td>2 D Math input mode displays the textbook-like version of what you input. Maple's simplification automatically combined fractions and places things in lowest terms.</td>
</tr>
<tr>
<td></td>
<td>49</td>
<td>(2.43)</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>$5!$</td>
<td>Use ! for factorial</td>
<td>Do you know what $5!!$ (double factorial) means?</td>
</tr>
<tr>
<td></td>
<td>120</td>
<td>(2.44)</td>
</tr>
</tbody>
</table>

Making mistakes

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2 + \left( \frac{3}{5} \right)$</td>
<td>Error message mistakes (from typos or mistakes in intentions)</td>
<td>The easiest ones to detect. You have to figure out what you are doing wrong, though. The error message may not always be helpful in advising you on this, although it often is.</td>
</tr>
<tr>
<td>$2 + \left( \frac{3}{5} \right)$</td>
<td>&quot;Logic errors&quot;</td>
<td>You are asking Maple to compute something that it understands, so it gives you an answer. However, this answer doesn't really solve your problem. You need to find a more appropriate computation, which you can only do by thinking about whether you went wrong. Often you can find these kinds of mistakes by looking at simpler versions of the problem where the answer can be figured out with paper and pencil. Then you can &quot;scale up&quot; the answer to handle the actual problem you have.</td>
</tr>
<tr>
<td>A farmer plants a fence post every foot, for 1250 feet. At that point, he switches to planting the fence posts every 4.7 feet for another 940 feet. How many fence posts does he need in all?</td>
<td>Maple did do the arithmetic in the above calculation correctly. The problem is that it's the wrong calculation. Do you see how to get the right answer?</td>
<td>The correct answer is $1251 + 201 1452$ fence posts. The computer did what it was asked to do -- the problem was that it was asked to do the wrong thing.</td>
</tr>
<tr>
<td></td>
<td>1250</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\frac{940}{4.7}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1450</td>
<td>(2.45)</td>
</tr>
</tbody>
</table>
Editing (fixing mistakes)

- backspace, delete erase starting from current cursor selection
- Arrow keys → ← move cursor within current selection
- Select with mouse/type replaces selected text
- Cut, copy and paste of a selection works as it does with a text processor

File saves, opens

- Save files with File -> Save or File -> Save As. Open a saved file with File -> Open. Other File operations similar to that of standard word processors.

Functions

\[
\sqrt{\csc \left( \frac{\pi}{2} \right)} + e
\]

\[
(1 + e)^{1/3}
\]

\[
\ln(e^2 \cdot \sqrt{e}) \quad \text{simplify symbolic} \quad \frac{5}{2}
\]

Insert math into an expression by using the Expression and Common Symbols Palette. If you are entering a function by the keyboard rather than the palette, you must enclose the function's argument in parentheses.

Chapter 2 demonstrated the following functions and symbols

<table>
<thead>
<tr>
<th>Square roots, ( n )-th roots</th>
<th>natural logarithms ( \ln ) (base ( e ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trig functions: ( \sin ), ( \cos ) (trig functions all use radians, not degrees)</td>
<td>Base 10 logarithms</td>
</tr>
<tr>
<td>( \arcsin ), ( \arccos ), ( \arctan )</td>
<td>sec, cosec</td>
</tr>
<tr>
<td>summation</td>
<td>( \pi ), ( e )</td>
</tr>
</tbody>
</table>

Algebra

\[
x^2 - 2x - 15 = 0 \quad \text{left hand side} \quad x^2 - 2x - 15
\]

\[
(x + 3)(x - 5)
\]

Right-click (control-click on Mac) on an entered expression to get the pop-up menu. Chapter 2 demonstrated examples of the following operations:

<table>
<thead>
<tr>
<th>factor</th>
<th>solve</th>
</tr>
</thead>
<tbody>
<tr>
<td>solve numerically</td>
<td>right hand side (of an equation)</td>
</tr>
<tr>
<td>left hand side (of an equation)</td>
<td>select (n-th part) of an expression</td>
</tr>
<tr>
<td>approximate numerically (to 5, 10, 20, etc. digits' accuracy)</td>
<td>plot (two dimensional) -- many plot options to determine range and domain of plot, color, captions, etc.</td>
</tr>
</tbody>
</table>

Evaluate at a point (choose values for variables in an expression)

Plotting

\[
sin(x) = 1 \quad \text{solve} \quad \left\{ x = \frac{1}{2} \pi \right\}
\]

\[
sin(x) = 1 \quad \text{solve} \quad 1.570796327
\]

\[
x^2 - \arcsin(x)
\]
The expression must be something that involves a single variable and will result in a number when a value is used for that variable. Thus you can't do a 2d plot of \( x^2 - a \) because you wouldn't get a number if you picked a value just for \( x \) (or just for \( a \)).

Maple uses defaults for the plot range, and the plot color.

Trying to plot an equation produces an implicit plot (see next appendix).

Plots->plot builder -> 2d plot

A dialog box appears that allows you the select from many more options, such as plot color, the line style, vertical and horizontal ranges, captions, etc.

Limited precision (decimal point) numbers

- \( \cos(x^2) = \sqrt{x} \) \( \text{solve} \quad 0.7352027350 \)
- \( 1 + \frac{2}{3} + \tan(1) + \pi^2 \)
  \( \text{solve} \quad 0.7666666667 + \tan(1) + \pi^2 \) (2.47)
- \( 2.3240743913549022305 + 3.1415926535897932385^2 \) (2.48)

Use of limited precision numbers in algebra (e.g. factoring, differentiation, solving) may not produce good results.

Use them in Maple only when an approximate result is desired.

Numbers like .25 or .6015 are limited precision. If you want exact algebra done, use 1/4 or 6015/10000, etc.

In very large calculations, limited precision calculations may be noticeably faster than those with exact arithmetic. Most of the time there isn't an appreciated difference.

Evaluate at a point

Example

\( x^2 - 2 \cdot a \cdot x = 0 \) \( \text{evaluate at point} \quad \frac{1}{4} - a = 0 \)

\( x^2 - 2 \cdot a \cdot x = 0 \) \( \text{evaluate at point} \quad x^2 - 6 \cdot a^2 \cdot x = 0 \)

\( 3 \cdot y + 5 \) \( \text{evaluate at point} \quad 14 \)

This operation will give a pop-up menu that will allow us to choose values for all the variables. In the first example, we we picked 1/2 for a value of \( x \). Note that the pop-up menu will show what you typed rather than displaying 2D math. In the second example, we specified "3*y^2" as the value for \( a \). In the third example, we picked 3 as the value for \( y \).
right hand side, left hand side
\[ x = \frac{\sin(a)}{p^2 - 1} \rightarrow \sin(a) \]
\[ x = \frac{\sin(a)}{p^2 - 1} \rightarrow x \]

Operations on multi-part expressions

<table>
<thead>
<tr>
<th>select entry</th>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x^2 - 4 \cdot x = 4 )</td>
<td>solve for ( x )</td>
<td>Solving this quadratic equation reveals that there are two solutions. Right clicking on these selections and then select entry (-1) produces the first solution. We can then approximate it by using the</td>
</tr>
<tr>
<td>([x = 2 + 2 \sqrt{2}], [x = 2 - 2 \sqrt{2}])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>([x = 2 + 2 \sqrt{2}], x = 2 + 2 \sqrt{2}, x = 4.8284)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Enter an expression in a document, then right-click (control-click on Mac) followed by:

<table>
<thead>
<tr>
<th>Operations on symbolic expressions</th>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>solve-&gt;solve</td>
<td>( x^2 - 1 ) solve</td>
<td>( (x = 1), (x = -1) )</td>
</tr>
<tr>
<td>solve-&gt;solve for a variable</td>
<td>( x^2 - 2 \cdot a \cdot x = 0 ) solve for ( x )</td>
<td>( [x = 0], [x = 2a] )</td>
</tr>
<tr>
<td>solve-&gt;numerically solve</td>
<td>( x = \cos(x) ) solve</td>
<td>( 0.7390851332 )</td>
</tr>
</tbody>
</table>

Factoring

\( x^2 - 1 \) factor \( (x - 1)(x + 1) \)
\( \cos(x)^2 - \sin(x)^2 \) factor \( (\cos(x) - \sin(x))(\cos(x) + \sin(x)) \)

Factoring can simplify an expression sometimes. Factoring doesn't know the trig simplification rules, though.

Plots->2d plot

\( x^2 - 1 \) →

The expression must be something that involves a single variable and will result in a number when a value is used for that variable. Thus you can't do a 2d plot of \( x^2 - a \) because you wouldn't get a number if you picked a value just for \( x \) (or just for \( a \)). Maple uses defaults for the plot range, and the plot color.
A dialog box appears that allows you the select from many more options, such as plot color, the line style, vertical and horizontal ranges, captions, etc.

<table>
<thead>
<tr>
<th>Operations on equations</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>right hand side, left hand side</td>
<td>( x = \frac{\sin(a)}{r^2 - 1} ) to ( \frac{\sin(a)}{r^2 - 1} ) move to right, move to left</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operations on constant expressions</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>approximate-&gt;5 (or 10, 20, 50)</td>
<td>( \tan\left(\frac{\pi}{10}\right) \sqrt{\frac{1}{10}} ) at 20 digits ( 0.34157868529293212152 ) uses approximation methods to turn constants into an approximate (decimal point) number with 5 (10, 20, 50) digits.</td>
</tr>
</tbody>
</table>
3 Chapter 3 Exact and limited-precision (floating point) numbers

3.1 Chapter Overview

In Maple, there are two types of numbers, *exact numbers* (printed as whole numbers, and fractions), and *limited-precision numbers* (written as a number with a limited number of digits after a mandatory decimal point).

The arithmetic Maple does with exact numbers is completely accurate.

The arithmetic Maple does with limited-precision numbers (like that of calculators or other standard computer languages) gets rounded at each stage to the number of digits being used. If the correct answer requires more digits than that, then some inaccuracy occurs.

The differences between exact and limited-precision numerical arithmetic leads us to the following rules:

a) Exact arithmetic is necessary if you are doing analytical work, such as a theoretical calculation or calculus homework.

b) Limited-precision arithmetic is faster but sometimes less accurate. It's good for situations where the numbers you're working with come from limited-precision measurements, and when you know you aren't going to get into trouble from the rounding errors made by keeping things in limited precision.

3.2 Exact arithmetic with exact numbers

There are three types of exact numbers in Maple, integers, fractions, and symbolic constants. If you enter an arithmetic expression with such numbers, Maple will do all the arithmetic exactly. It will also automatically evaluate functions that have simple exact results, such as

$$\sin \left( \frac{\pi}{6} \right) = \frac{1}{2}.$$ 

Some exact numbers

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integers (whole numbers)</td>
<td>2, 5, -1005</td>
<td>In Maple, integers can be up to hundreds of thousands of digits long.</td>
</tr>
<tr>
<td>Fractions (ratios of integers)</td>
<td>3/5, 9/7, 14597</td>
<td>In Maple, the numerator and denominator can be up to hundreds of thousands of digits long.</td>
</tr>
<tr>
<td>Symbolic constants</td>
<td>π, Pi, e, exp(1), (\sqrt{2}), sqrt(2), sin(60), (\infty), infinity, (\sqrt{-1}, I)</td>
<td>You can enter (\pi) from the Common Symbols Palette, or you type the letters Pi from the keyboard to get the same thing. The first letter must be capitalized. You can enter (e) from the Common Symbols Palette, or you can type the letters exp(1) from the keyboard. Typing (e) from the keyboard does not work -- it means the algebraic unknown &quot;e&quot; (as in &quot;x&quot;, &quot;y&quot;, etc.), not a constant. You can enter (\sqrt{2}) from the Expression Palette, or you can type the letters sqrt(2) from the keyboard. While entering (\sin(0)) using the Expression Palette would return a result of 0, entering (\sin(60)) results in the same thing. There is no simpler way of expression this constant. Maple always uses radians for trig functions.</td>
</tr>
</tbody>
</table>
so this is not talking about the sine of 60 degrees, but rather the sine of 60 radians.

You can enter \( \infty \) from the Common Symbols palette, or infinity by typing on the keyboard.

Maple uses “capital I” as the symbolic constant the square root of -1. Select \( i \) from the Common Symbols palette, or type shift-i from the keyboard. Lower case i (i without the shift key) is not a symbolic constant.

Arithmetic with exact numbers is always exact. If there is no exact result for an expression, Maple will leave things alone.

**Exact arithmetic is always exact**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 3 + \frac{5}{7} + \frac{9}{11} + \frac{13}{15} )</td>
<td>( \frac{6236}{1155} )</td>
</tr>
<tr>
<td>( \tan\left(\frac{30\pi}{28!}\cdot\text{Pi}\right) + \ln\left(\text{e}^{-\frac{1}{10}}\right) )</td>
<td>( -\frac{1}{10} )</td>
</tr>
<tr>
<td>( \tan\left(\frac{30\pi}{28!}\cdot\text{Pi}\right) + \ln\left(\text{exp}\left(-\frac{10}{3}\right)\right) )</td>
<td>( \frac{10}{3} )</td>
</tr>
<tr>
<td>( \tan(870\pi) - \frac{1}{10} \ln(\text{e}) )</td>
<td></td>
</tr>
<tr>
<td>( \sqrt{\frac{35}{15}} )</td>
<td>( \frac{8575}{3} )</td>
</tr>
</tbody>
</table>

Maple takes fractions and integers and produces the integer or fraction that results. It will automatically divide out the common divisor of numerators and denominators.

We did this by entering \( \sqrt{\frac{35}{15}} \) into a document box, and then typing control-= (control key and equals key at the same time.)

**3.3 Limited-precision (calculator-style) arithmetic with limited-precision numbers.**

If you see a number on your Maple document with a decimal point, it's a **limited precision number** or **floating point number**. If you see a number without a decimal point, it's an exact number. Maple performs calculator-style arithmetic (called limited-precision arithmetic or floating point arithmetic) when it sees this number.

By default, Maple creates limited precision numbers with ten decimal digits' accuracy. However, if you enter a number with a decimal point that has more digits than that, it will create it with as many digits as you use. Limited precision numbers can be written in scientific notation using an "e" to indicate the exponent. You can't use "e" syntax to enter an exact number.

Limited-precision numbers are sometimes called **floating point numbers**.

**Limited-precision (floating point) numbers**

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>Note that this is not the same as the exact number zero (0)... it has a decimal point after it.</td>
</tr>
</tbody>
</table>
If you enter a number that needs more than ten decimals, Maple will use them all.

At some point, Maple will display a large or small number using powers of ten rather a lot of leading-zero decimal places.

You can type in a number with a power of ten using the "e" notation. Here "e" is the symbol typed from the keyboard, not the e from the Common Symbols keyboard.

You can also use the capital E typed from the keyboard.

| 0.00          | 0.          | (3.2) |
| .12345678901234567890 | 0.12345678901234567890 | (3.3) |
| -.00001367890 | -.00001367890 | (3.4) |
| -.0000000000001367890 | -.1367890 \times 10^{-13} | (3.5) |
| 1.69e35      | 1.69 \times 10^{35} | (3.6) |
| 0.00169E-22  | 1.69 \times 10^{-22} | (3.7) |

Calculator-style arithmetic differs from exact arithmetic in that the computer uses only a fixed number of decimal places (by default, ten). If more than that is needed to write down the answer, the computer will round the result back to the fixed precision (number of decimal places) being used. This may result in relatively small error. However we shall see situations where the rounding error can contaminate the accuracy of the result, to the point where it can't be believed.

**Limited-precision (floating point) arithmetic**

| \[1 + .1 + .02 + .9e-9 + .8e-12\] 1.120000001 | (3.8) |
| \[1 + .1 + .02 + .9e-9 + .8e-12 - 1 - .1\] 0.020000001 | (3.9) |
| \[1 + .1 + .02 - .1 - 1 + .9e-9 + .8e-12\] 0.0200000090 | (3.10) |

We see that the result has rounded away some of the smaller terms.

We've subtracted away 1.1, but only after the rounding has occurred for the intermediate steps of the computation. So we've lost the smaller digits.

We get a more accurate result if we add the numbers together in a different order. This means limited-precision arithmetic does not obey the law of commutivity of addition.

| \[1.0 \div 3.0\] 0.3333333333 | (3.11) |
| \[1.0 - 1.0 \div 3.0\] 1.10^{-10} | (3.12) |

Because of rounding, the result of dividing one by three is not exactly one-third.

Maple's exact result is exactly correct.
By using a floating point number, Maple computes an approximation to \( \sin(\pi) \). It's not quite the same as the exact result.

\[
\begin{align*}
\sin(3.1415926535) &= -4.102067615 \times 10^{-10} \\
\sin(\pi) &= 0
\end{align*}
\]

This means (the limited-precision version of) \( 25999874 \times 10^5 \).

This means (the limited-precision version of) \( 3 \times 10^{-35} \). Note that Maple does not display the "." for multiplication between the number and the power of ten. Note that this situation is one of the few times where we can enter a limited precision number without a decimal point. This is because of the "e".

This is an exact number, not a limited precision number. Note that there are no decimal points used in the input, so Maple considers this to be an arithmetic operation involving only exact numbers. It's how we enter \( 3 \times 10^{-35} \) as an exact number.

The result is expressed as a limited-precision number in scientific notation because it is large, even though we didn't enter any numbers that large.

### 3.4 Maple will decide what kind of arithmetic to use by how numbers look

First note that any "calculator number" can be written as a fraction. For example, 1.98376 can be written (somewhat less conveniently) as \( 198376/10000 \). Therefore, if we want to talk about any (real) number, we can use an integer, fraction, or symbolic constant. Like most computer programming languages, Maple processes only what it sees written. It can't read your mind that you really meant \( \pi \) when you entered 3.1415, or that you meant \( 1/3 \) when you write .3333333333. Maple follows this rule:

if you write a number with a decimal point in it, Maple treats it as a limited-precision number and does limited-precision arithmetic with it

if you write a number without a decimal point in it, Maple treats it as exact and does exact arithmetic with it. It will also perform algebraic simplifications exactly, such as

\[
\sin\left(\frac{\pi}{6}\right) = \frac{1}{2}
\]

If you mix two both kinds of numbers, Maple will eventually start using floating point arithmetic when it encounters the floating point numbers.
3.5 Differences between limited-precision numbers in Maple and in math text books

There are some subtle differences between "the mathematical world", "the math student world", and "the Maple world" in how numerical notation works.

12. Math textbooks and students use ".1" and "1/10" interchangeably. Maple regards them as completely different. Because of the "contagion" of limited-precision arithmetic, if you want an all exact calculation (because the problem requires you to express the answer using fractions, for example), then you cannot use .1 in the expression you enter into Maple.

13. Students sometimes use numerical approximations as being the same number as what they approximate. For example, some people start believing that "3.14" or "3.14159265" is the same as π, or that .3333333333 means the same thing as 1/3. Maple knows better. You'll get wrong answers if you try to push calculator expectations into a computer environment where the computer follows the rules of algebra, as illustrated by the example ???.

14. Factoring works for polynomials with exact coefficients, but not necessarily for polynomials using approximations for the fractions. It would probably work better if we were using numbers like ".25" or "\.6061" which require fewer than ten digits to write down with complete accuracy.

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x^2 - \frac{10}{3}x + \frac{49}{16} - \frac{41}{144} \quad \text{factor} \quad \frac{1}{9}(3x - 5)^2</td>
<td>Maple can factor polynomials with exact coefficients, if there is a way of factoring them with integer or rational numbers.</td>
</tr>
<tr>
<td>x^2 - 3.333333333x + 2.777777778 \quad \text{factor} \quad x^2 - 3.33333332x + 2.777777777</td>
<td>Maple's factor operation can't find factors for polynomials that only approximate the exact coefficients.</td>
</tr>
</tbody>
</table>

15. Numerical solving doesn't necessarily produce the same results when approximations are used for a polynomial's coefficients. Here the use of decimal point approximations for the coefficients, even though they are about as precise as ten digits can handle, produce complex number roots rather than real roots! They are "close", but they are they qualitatively different.

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>{x = \frac{5}{3}, x = \frac{5}{3}}</td>
<td>When we right-click-&gt;solve for variable x, we see that the equation has a double root.</td>
</tr>
<tr>
<td>1.666666667, 1.666666667</td>
<td>When we right-click-&gt;solve-&gt;numerically solve, we see approximations to the two roots. Some amount of exact arithmetic is tried initially to see if there are exact values to be found. Then these values are numerically approximated.</td>
</tr>
<tr>
<td>(x = 1.666666666 + 0.00015092308561), (x = 1.666666666 - 0.00015092308561)</td>
<td>When we approximate the coefficients with floating point numbers, and then solve, all the solving is done with floating point numbers. We see that the solver (which is trying as best as it can given that it can only work with limited precision), instead of a double root, it finds two complex number roots -- &quot;I&quot; is Maple's symbol for the square root of -1. This is an example where wholesale use of floating point arithmetic produces results that fall surprisingly short of the mathematical truth.</td>
</tr>
</tbody>
</table>

Use of limited-precision numbers can produce results that differ in surprising ways from the way we think of numbers behaving. For example, if you add four floating point numbers in two different orders, you may get different results!
### Contrasting approximate and exact arithmetic

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \frac{1}{1000} + 20000000000 + 1 - 20000000000 - 1 ]</td>
<td>We add together four numbers. We get a result.</td>
</tr>
<tr>
<td>[ \frac{1}{1000} ]</td>
<td>We add together the four numbers in a different order. We get the same result because &quot;addition is commutative&quot;.</td>
</tr>
<tr>
<td>[ 20000000000 + 1 - 20000000000 - 1 + \frac{1}{1000} ]</td>
<td>We do this with limited precision numbers. We don't get anywhere near the same answer. In fact, we get almost as many answers as ways we have of writing the expression. This would probably be true of your pocket calculator, if it kept only 10 digits of precision. You could manufacture an example that would show something similar happening no matter how many digits your calculator kept.</td>
</tr>
<tr>
<td>[ 1.0e-3 + 2e11 + 1.0 - 2e11 - 1.0 ]</td>
<td>The bottom line: with limited-precision numbers, evidently ( a + b + c = a + c + b ) is not always true. What other kinds of mathematical laws does your calculator disobey? If this does not make you nervous, then you'd probably be fine with an engineer who designed a supersonic airplane using &quot;not exactly&quot; the laws of airflow or built a chemical refinery using &quot;not exactly&quot; the laws of chemistry!</td>
</tr>
<tr>
<td>[ -1.0 ]</td>
<td></td>
</tr>
<tr>
<td>[ 2e11 + 1.0 - 2e11 - 1.0 + 1.0e-3 ]</td>
<td></td>
</tr>
<tr>
<td>[ -0.9990 ]</td>
<td></td>
</tr>
<tr>
<td>[ -2e11 + 1.0 - 1.0 + 1.0e-3 + 2e11 ]</td>
<td></td>
</tr>
<tr>
<td>[ 0 ]</td>
<td></td>
</tr>
<tr>
<td>[ 2e11 - 1.0 - 2e11 + 1.0 + 1.0e-3 ]</td>
<td></td>
</tr>
<tr>
<td>[ 1.0010 ]</td>
<td></td>
</tr>
<tr>
<td>[ -2e11 - 1.0 + 1.0e-3 + 2e11 + 1.0 ]</td>
<td></td>
</tr>
<tr>
<td>[ 1.0 ]</td>
<td></td>
</tr>
</tbody>
</table>

This gives us a few rules of thumb for using numbers with decimal points in them. a) If you are doing algebra (e.g. "deriving a formula"), don't use .1, .75, or .206, use 1/10, 3/4, or 206/1000. instead. b) If the problem you are working needs only approximate answers (e.g. "using a formula"), then enter a formula with all-exact numbers, do any algebra necessary, and then switch to numerical solving or approximations. c) Only if you are sure that you can perform all the steps with numerical arithmetic and get an answer accurate enough for your purposes should you start off in numerical mode. If you need to do algebra, you should stick with exact numbers until you want to stop doing algebra and just do calculation.

In some technical systems, such as Matlab or Python, the primary kind of numbers are integers (up to a fixed size, such as 9 digits), and limited-precision (usually with 16 digits accuracy maximum). Exact numbers are either completely unavailable, or they are available only through means that require much more extra work (more typing, usually). Maple is one of the few systems where both kinds of numbers exist together.

We observe that when some technical people use limited-precision numbers, they usually skip the accuracy testing and just hope that the limited precision isn't going to bite them since much of the time the computed results are reasonably accurate. Unfortunately this is not always the case, especially when it comes to situations involving large amounts of money. See for example http://books.google.com/books?id=NDbQe52pX3kC&pg=PA38&lpg=PA38&dq=Vanouver+stock+exchange+roundoff+error&source=web&ots=vkseL01hBP&sig=FBXWmcpsAqq4f37Zz7cvQOm-ILZM&hl=en&sa=X&oi=book_result&resnum=2&ct=result#PPA38,M1 or page 183 of http://www.validlab.com/goldberg/paper.pdf. The person doing the calculations is always responsible for their appropriate use. If you are in a high-stakes situation with your professional or academic reputation on the line, the only responsible path is to check your limited-precision work for its (in)-accuracy.
### 3.6 Which kind of numbers and arithmetic should I use?

Exact arithmetic makes it easy to do formula manipulation (algebra) with the computer correctly, but limited precision arithmetic is often faster. So,

<table>
<thead>
<tr>
<th>Guidelines for using exact and limited-precision numbers in Maple</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. If you're doing a calculation for mathematical or theoretical analysis and are given exact values for the parameters of the problem, use integers, fractions, and symbolic constants in your expressions. If get a number at the end of the analysis, and you want &quot;ballpark&quot; estimates of the magnitudes, you can always approximate these exact numbers at the end of the computation. <strong>IN THIS SITUATION AVOID USING NUMBERS WITH DECIMAL POINTS -- use 1/2 instead of .5, 1/10 instead of .1, Pi instead of 3.1415.</strong></td>
</tr>
<tr>
<td>2. If you're doing a non-analytical computation (e.g. a computation where all the numbers are taken from measurements, not from theory) then it's probably okay to use numbers with decimal points. As we have seen, even if you mix exact and limited-precision numbers, most of the calculation will probably be done in limited precision.</td>
</tr>
</tbody>
</table>

### 3.7 Summary of Chapter 3 material

They all have a decimal point or an eXX in them.

#### Examples of floating point (limited precision) numbers

- \( 1.0, 2.5e-3, 23.0e-1, -999667.3e2, 9876e53 \)
- \( 1.0, 0.0025, 2.30, -9.996673 \times 10^7, 9.876 \times 10^{56} \) (3.25)

#### Examples of exact numbers -- integers and rational numbers (fractions)

- \( 111222, \frac{11}{33} \)
- \( 111222, \frac{1}{3} \) (3.26)

#### Examples of exact numbers -- symbolic constants

- \( \pi, \infty \)
- \( \pi, e, 1, \infty \) (3.27)

### Differences between math world and Maple world

<table>
<thead>
<tr>
<th>Math world</th>
<th>Maple world</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{1}{10} ) and .1 mean the same thing</td>
<td>1/10 is an exact number, .1 is a floating point number. The most important difference between them is that an arithmetic expression with all exact numbers will use exact arithmetic, whereas an expression with all floating point numbers will use limited precision arithmetic, which is different.</td>
</tr>
</tbody>
</table>
| sin(\( \pi/3 \)) and sin(\( \pi/3.0 \)) mean the same thing | Exact identities don't apply necessarily to floating point numbers. sin\( \left( \frac{\pi}{3} \right) \), sin\( \left( \frac{\pi}{3.0} \right) \)
\[
= \frac{1}{2} \sqrt{3}, \sin(0.3333333333 \pi)
\] (3.28) |
| factoring and other algebraic operations work on any expression | algebraic operations are only guaranteed to work well on expressions with all exact numbers. |
| Operations on all numbers obey basic laws such as commutativity of addition | Operations on exact numbers obey basic laws such as commutativity of addition. Arithmetic on floating point (limited precision) numbers do not obey the commutative law of addition. (This is true not only for Maple but for calculators and for the arithmetic hardware of most computers.) |

### Why there are two kinds of numbers in Maple
<table>
<thead>
<tr>
<th>Exact arithmetic is completely accurate, but somewhat slower than 10 digit floating point arithmetic.</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 digit floating point arithmetic can be faster compared to doing calculations with very large fractions or integers, but is not completely accurate</td>
</tr>
</tbody>
</table>
4 Chapter 4 Words, labels, assignments, and scripts

4.1 Chapter Overview

We learn how to use Maple as a word processor. This allows us to "write up" computations with explanation being provided along the way.

We learn how to use the labels that Maple uses for each expression that we enter into a worksheet by hitting the enter (on some computers, return) key. They can be useful to retrieve computed values in subsequent steps of a multi-step computation.

We learn how to use the assignment operation := to label results with symbolic names. Like labels, these names can be used in subsequent steps of a computation. Assignment is a fundamental feature of used in writing computer programs.

4.2 Maple as a word processor

Maple documents allow a mixture of text and mathematics. Here's a short description: 1. Position the cursor at the spot where you want to enter text.

2. Type control-T (on Macintosh, command-T). Alternatively, you can use the Maple menu Insert->Text. The Toolbar will show that you are text mode.

3. With the keyboard, enter your verbiage. When you are in text mode, you will also see that the menu bar will be enabled for boldface, italics, underlining, left-/center-/right-justified text, colored text, and colored backgrounds, and bulleted/numbered text.

4. To switch back to math, type control-R (on Macintosh, command-R). Alternatively you can use the Maple menu Insert->2-D Math.

Document after control-T (or Insert->Text).
Document with a mixture of text and math

Richard saw the following equation in his calculus textbook. It described Newton's law of gravitation in these terms: \( F = \frac{G m_1 m_2}{r^2} \). He had seen the law written in other terms: \( f = \frac{m_1 m_2}{r} \).

The user typed the text, then did control-R (command-R) and entered the \( F = G \ldots \) formula in math mode. The user got the subscripted \( m_1 \) by typing an underscore _ after the \( m \) to get Maple to descend to subscripts, then used the right-arrow key to ascend back up to the main level of the expression. A similar alternating between text mode and math mode allowed entry of the second sentence. No use of underscores was necessary in the second sentence because \( m_1 \) and \( m_2 \) aren't written with any subscripts.

Note that Tools->Spellcheck (alternatively, the F7 key) will run a spelling check on the non-math part of your document.

It is possible to mix text and clickable calculations in a paragraph. Typing control-= (command-=) when the cursor is in a math expression will cause Maple to print an "=" and then the result of evaluating the expression on the same line. This is an alternative to hitting the enter key and allows those kinds of calculations to be mixed with text.

control-= puts the results of a calculation in the midst of text.

The user entered text, then did a control-R (command-R), then entered the math expression and then typed control-=. After the calculation result appeared, they typed control-T and entered the remainder of the sentence.

### 4.3 Labels and assignments: remembering results for future use.

#### Labels

Whenever you enter an expression and then hit enter you are telling Maple, "Evaluate this expression (do any arithmetic or function calculations it specifies) and print out the result you get". The results are automatically labeled by Maple, given a bold face number (of the form \( X \) or a segmented label such as \( XX.YY.ZZ \)). You can refer to the results by the Maple menu Insert->Label and then typing in the number. The keyboard shortcut for this is control-L (command-L on Macintosh).

#### Labeled results

<table>
<thead>
<tr>
<th>Maple work</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>5!</td>
<td>Compute a result</td>
</tr>
<tr>
<td>4!</td>
<td>Compute another result</td>
</tr>
<tr>
<td>120</td>
<td>Add them together</td>
</tr>
</tbody>
</table>

(4.1) Find the ratio of the two results.
Assignment to a name, evaluation

The Maple operator `:=` (colon immediately followed by an equals) assigns a value to a name. After that point in time, if you do a calculation with an expression that uses the name, the assigned value is used.

When you enter an expression in Math mode (even if it's just a name, without any arithmetic), you ask Maple to evaluate what you enter. Maple also (before it does anything else) figures out the values assigned to names that appear in the expression. Then it does arithmetic, function calculation, and any other operations you've described in the expression.

### Assignment

<table>
<thead>
<tr>
<th>Examples of assignment with <code>:=</code></th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p := x^2 + x + 3 x + 5 )</td>
<td>We assign the name ( p ) the value of the expression.</td>
</tr>
<tr>
<td>( p + 1 )</td>
<td>(4.5) If we enter an expression containing ( p ), its value is plugged in for the calculation of the result.</td>
</tr>
<tr>
<td>( x := 3 )</td>
<td>(4.6) Here we assign the name ( x ) the value 3.</td>
</tr>
<tr>
<td>( p )</td>
<td>(4.7) If we now do a calculation with ( p ) (here all we ask is for Maple to calculate the current value of ( p )), the value of ( x ) is used since ( p )'s value mentions ( x ). Thus there may be a chain of assignments that Maple must look at.</td>
</tr>
<tr>
<td>( x := 4 )</td>
<td>(4.8) Solving the result 1.1.2.4 for ( a ) can be gotten by right clicking that expression.</td>
</tr>
<tr>
<td>( p )</td>
<td>(4.9) We change the value of ( x ) by assigning it a different value.</td>
</tr>
<tr>
<td>( { a = - \frac{17}{3} } )</td>
<td>(4.10) When we do another calculation with ( p ), the most recent assigned value of ( x ) is used.</td>
</tr>
<tr>
<td>( x := 4 )</td>
<td>(4.11) We can undo the connection between ( x ) and any value by unassigning ( x ). This operation produces no output, so no label. We can barely tell that it has happened.</td>
</tr>
<tr>
<td>( p )</td>
<td>(4.12) ( p ) still has a value, but since ( x ) no longer has a value, we are back to the original result.</td>
</tr>
<tr>
<td>( \text{unassign('x')} )</td>
<td></td>
</tr>
</tbody>
</table>
Assignment is a feature common to many programming languages. In some languages "\(\text{:=}\)" is used for the assignment operation. In others plain equals "\(\text{=}\)" is used. Maple uses "\(\text{:=}\)" because it uses "\(\text{=}\)" for equations. It would be confusing to use the same symbol for two common but different operations.

\(\text{=}\) and := mean different things in Maple

**Example**

<table>
<thead>
<tr>
<th>(a := 3)</th>
<th>We assign (a) the value 3.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x := 4)</td>
<td>This is an equation. It doesn't assign (x) any value.</td>
</tr>
<tr>
<td>(p := a + x)</td>
<td>We assign (p) the value of the expression (a + x). (a) stands for the value 3 at this point since we did an assignment to it. (x) is just a symbol that has no assigned value.</td>
</tr>
<tr>
<td>(x := 47)</td>
<td>We can do an assignment to (x).</td>
</tr>
<tr>
<td>(p := a + x)</td>
<td>This time (p)'s value is 3 + 47.</td>
</tr>
</tbody>
</table>

### 4.4 The structure of information in Maple: getting information from solve

The result of the `solve` operation can have multiple parts if there are multiple solutions to the equation. We can select each part by giving an index (either 1 or 2).

\[eq1s := 3 \cdot x = x^2 - 28 \Rightarrow \{x = -4\}, \{x = 7\}\]

\[eq1s \Rightarrow 3 \cdot x = x^2 - 28 \Rightarrow \{x = -4\}, \{x = 7\}\]

If we give `solve` a linear equation, it has only one solution. We can still select the first entry.

\[eq2s := 3 \cdot x = 28 \Rightarrow x = \frac{28}{3}\]

If we do "solve for \(x\)" for the same equation, we see that the answer comes back in a slightly different form. But it still has parts.

\[eq1L = 3 \cdot x = x^2 - 28 \Rightarrow [x = -4], [x = 7]\]

Maple (as well as many other programming languages) can compute with objects that have structure. Here are three different kinds of structures that Maple can handle:
### Basic data structures in Maple

<table>
<thead>
<tr>
<th>Type of structure</th>
<th>What they look like</th>
<th>Examples</th>
<th>Access (Select-&gt;i on right-click/control-click menu also works for access)</th>
<th>Empty structure value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sequences</strong></td>
<td>Values separated by a comma</td>
<td>19, 47, 92</td>
<td>(1.4.3)[1]</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>19, 47, 92</td>
<td>(1.4.3)[ -1]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>92</td>
<td>(1.4.3)[ 1, 2]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>19, 47</td>
<td>(1.4.3)[ 1, 2]</td>
<td></td>
</tr>
<tr>
<td><strong>Lists</strong></td>
<td>A sequence surrounded by square brackets [ ]</td>
<td>MyData := [1.0, x, 3/4, a]</td>
<td>MyData[2]</td>
<td>[ ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[1.0, x, 3/4, a]</td>
<td>MyData[5]</td>
<td>[ ] is the empty list.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Error, invalid subscript selector</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MyData[2, 4]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[x, 3/4, a]</td>
<td>(4.27)</td>
</tr>
<tr>
<td><strong>Sets</strong></td>
<td>A sequence surrounded by curly braces { }</td>
<td>Scores := {3, 7, 10}</td>
<td>Scores[1]</td>
<td>{}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{3, 7, 10}</td>
<td>Scores[1]</td>
<td>{}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Scores[ -3]</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Scores[ -4]</td>
<td>3</td>
</tr>
</tbody>
</table>

For the time being, we just want you to recognize the different kinds of structures that are output by *solve* and other functions and be able to select parts from them. Later on we will get a lot of work done by performing computations with them.
4.5 Scripting: creating computational work in reusable form

Consider the problem you did in Lab 1, along with a solution:

Version 1 and solution

From Anton, *Calculus*, 8th edition, ch. 1 review exercises, problem 37, p. 99. A breeding group of 20 bighorn sheep is released in a protected area in Colorado. It is expected that with careful management the number of sheep, \( N \), after \( t \) years will be given by the formula:

\[
N = \frac{220}{1 + 10 \cdot (0.83)^t}
\]

and that the sheep population will be able to maintain itself without further supervision once the population reaches a size of 80. (a) Graph \( N \) versus \( t \). (b) How many years must the state of Colorado maintain a program to care for the sheep? Show your work. (c) How many bighorn sheep can the environment in the protected area support? (hint: examine the graph for large values of \( t \)).

(a) \( N = \frac{220}{1 + 10 \cdot (0.83)^t} \)

(b) \( N = \frac{220}{1 + 10 \cdot (0.83)^t} \) evaluate at point \( 80 = \frac{220}{1 + 10 \cdot 0.83^t} \)

(c) We do this by drawing the graph for a large enough value of \( t \) and seeing what happens.

\[
N = \frac{220}{1 + 10 \cdot (0.83)^t} \text{ right hand side } \rightarrow \frac{220}{1 + 10 \cdot 0.83^t}
\]
Alternatively (this is optional) we can do a little calculus and take the limit of the expression as \( t \) goes to infinity.

\[
N = \frac{220}{1 + 10 \cdot (0.83)^t} \quad \text{right hand side} \quad \frac{220}{1 + 10 \cdot 0.83^t} = 220.
\]

We can imagine ourselves working as an environmental engineer for the National Forest Service and being very pleased with ourselves for solving the problem with Maple. But now we are handled two more problems to do:

**Version 2**

A breeding group of 33 bighorn sheep is released in a protected area in Colorado. It is expected that with careful management the number of sheep, \( N \), after \( t \) years will be given by the formula:

\[
N = \frac{330}{1 + 10 \cdot (0.79)^t}
\]

and that the sheep population will be able to maintain itself without further supervision once the population reaches a size of 85.

(a) Graph \( N \) versus \( t \). (b) How many years must the state of Colorado maintain a program to care for the sheep? Show your work.
(c) How many bighorn sheep can the environment in the protected area support? (hint: examine the graph for large values of \( t \).)

**Version 3**

A breeding group of 45 bighorn sheep is released in a protected area in Colorado. It is expected that with careful management the number of sheep, \( N \), after \( t \) years will be given by the formula:

\[
N = \frac{450}{1 + 10 \cdot (0.63)^t}
\]

and that the sheep population will be able to maintain itself without further supervision once the population reaches a size of 90.

(a) Graph \( N \) versus \( t \). (b) How many years must the state of Montana maintain a program to care for the sheep? Show your work.
(c) How many bighorn sheep can the environment in the protected area support? (hint: examine the graph for large values of \( t \).)

We have the feeling that we will shortly be handed problems for a number of other locations as well. How can we reuse our original work with minimal effort?

If we had the first solution, we could produce the second solution through *copy-paste-edit-re-execute*:

**Executing a clone of a script through copy-paste**
1. Copy and paste the original solution into a new document, or to a spot later in the same document. You do this as with a word processor -- select a region of the worksheet with the mouse, type control-C (command-C on a Mac). Then position the document cursor in the destination, and type control-V (command-V on a Mac). A duplicate of the solution should appear there.

2. Using the mouse, edit the formulae as needed. You only need to edit the black (input) formulae, not the blue ones.

3. Position the mouse at the first computation and hit enter. Continue to work your way through the sequence of the commands.

4. Alternatively, select the entire region containing the edited version of the solution and hit Edit->Execute->Selection.

The results of executing the edited script are (page 46). It is not totally correct. We will have to change a few things by hand: the clickable operation "evaluate at a point" in step (b) will use N=80 (which is what the copy says) instead of N=85, so we will have to redo it. Also, the last plot will not show an appropriate vertical range unless we do that. We'd like to do a little less of this hand-tuning in re-execution. Also, we'd like to make it easier to remember what to change between multiple versions.

(page 47).

### 4.6 Rewriting the script using labeling and assignment

While copying and editing is probably a little faster than typing in the whole script again, we can reduce the amount of hunting around for changes by writing a script finding the parameters of the problem, and writing the script so that it assigns values to the parameters at the beginning of the problem.

**Finding and naming parameters**

You can find parameters if you have several versions of a problem by looking at what changes in the formulas from version to version. This only works if you have solved the at least one version of the problem first. For example, in the sheep problem, we note the following things changing. We pick names for these.

1. the numerator of the "sheep equation" (P)
2. the coefficient in the denominator of the equation (c)
3. the value of the stable population (s)

The other number in the script is the original population of the sheep. We realize that it can be derived by evaluating the "sheep equation" at t=0. Thus this value does not need to be a parameter as we can derive it from the other information. It might be a good check though. We then write the script to assign values to the parameters at the start of the script, and then write the other operations and expressions in terms of the symbols.

This allows us to redo the script just by changing the values at the beginning of the script. We use the word processing features to add extra directions to make the script easier to use, basically saying "here, change these things", and "this is the end of where you should stop copying".

The result of executing the script is (page 48).

Having created this script, we can handle the third version of the problem by editing the values of the parameters and executing the script again by selecting the entire script with the mouse, and then doing Edit->Execute->Selection. We get this result:

Note: a "bug" in Maple 13 causes duplicate graphs to sometimes appear during re-execution of the script. It's easy enough to delete the redundant copies.
4.7 Summary of script writing

Script writing is appropriate when you expect to handle several different versions of the same problem. In professional work, this is often the case -- if it's worth doing at all, it's probably going to occur more than once.

Don't try to write a script before you've figured out how to do the problem! It may be possible to tweak a script a bit if it doesn't work exactly right initially, but if you'll just be scripting garbage if your initial thoughts are far away from what you really want.

Once you have a worksheet of instructions for one version of the problem look at it and the other versions and find the parameters. Set up a new worksheet where the first thing you do is to assign values to the parameters. Then work through the rest of the instructions and rewrite them to use the parameters instead of the fixed values from one version of the problem.

4.8 Troubleshooting scripts

Programming books have the tendency to show things that work. They say less about what to do about the things that don't work. However, it is usually the case that computer users often spend more time "getting things to work" than "working". Learning how to get out of jams is at least as valuable as knowing how to enter operations and what they mean. We offer this advice, which is "commonsense about work" as applied to interactive software development:

1. Know how to solve the problem before you start scripting. If you have only fuzzy notions about the math or the operations you want to proceed, your computer work will just amplify that. Where does this plan come from? If you are lucky, the solution may be told to you. But the big bucks, as they say, go to those who can devise the solution plan themselves.

2. Break the actions into small steps. If you get into trouble, you will be able to nip it at the bud. This becomes a more prominent tactic in later work when we are tempted to construct long-winded one-line expressions that do everything at once.

3. Test pieces individually, then put them together. For example, if you don't really understand how to make a dotted line plot in teal with a title, you should try to do that with a simple plot (such as $x^2$) rather than an expression whose shape you aren't that familiar with it. Then take what works and substitute the real expression you want to plot in a copy of the $x^2$ plot operation. If you think about it, this is similar to what happened in Fall 2009 ENGR 101 Lab 2, where they first had you learn how to trigger an oscilloscope with the output of the function generator, rather than the output of the camera/flash sensor. The complexity of troubleshooting is reduced if you half as many unknowns to worry about. This can be called divide and conquer troubleshooting.

4. Match parentheses. If you're building an expression that has parentheses, make sure that the number of beginning parentheses is equal to the number of closing parentheses. You can do this by counting! If you have more ( than ), then look in the expression to see where the missing one might be. 5. If what you have doesn't work, find something similar which does work, and then incrementally edit it. For example, if you can't get $a := x^2 + 3\cdot x + 1$ in, then first see whether you can get $a := 1$ to work. Once you succeed with that, edit the expression to $a := 3\cdot x + 1$ and so forth.

6. If strange things continue to happen despite your best efforts to troubleshoot, it may be that previous settings in the Maple session are interfering with your current work. Recall that some people in the ENGR101 lab couldn't get their oscilloscopes to work because of settings change in the oscilloscopes by groups earlier in the day. This can be particularly true if you are developing scripts and are assigning parameter values, then switch to development of another script in the same document. The values you assigned will not magically unassign themselves when you start working on something new in the same worksheet. The remedy for this is: open a new Maple document (File->New->Document). If you really know what you are doing, you can use the unassign operation (first mentioned here), or the restart operation (an advanced command).
4.9 Attachments

Attachment: Version 2 of sheep script (slightly incorrect), without parameters

A breeding group of 30 bighorn sheep is released in a protected area in Colorado. It is expected that with careful management the number of sheep, \( N \), after \( t \) years will be given by the formula:

\[
N = \frac{330}{1 + 10 \cdot (0.79)^t}
\]

and that the sheep population will be able to maintain itself without further supervision once the population reaches a size of 85. (a) Graph \( N \) versus \( t \). (b) How many years must the state of Colorado maintain a program to care for the sheep? Show your work. (c) How many bighorn sheep can the environment in the protected area support? (hint: examine the graph for large values of \( t \).)

(a) \( N = \frac{330}{1 + 10 \cdot (0.79)^t} \)

(b) \( N = \frac{330}{1 + 10 \cdot (0.79)^t} \) evaluate at point \( t = 8 \):
\[
80 = \frac{330}{1 + 10 \cdot 0.79^8} = 4.934410722
\]

(c) 
\[
N = \frac{330}{1 + 10 \cdot (0.79)^t} \text{ right hand side } = \frac{330}{1 + 10 \cdot 0.79^8}
\]
Alternatively (this is optional) we can do a little calculus and take the limit (left-sided limit) of the expression as \( t \) goes to infinity.

\[
N = \frac{330}{1 + 10 \cdot (0.79)^t} \quad \text{right hand side} \quad \frac{330}{1 + 10 \cdot 0.79^t} \to 330.
\]

**Attachment: Version 2 of sheep script (corrected), without parameters**

Version 2 of sheep problem, with edited script

A breeding group of 30 bighorn sheep is released in a protected area in Colorado. It is expected that with careful management the number of sheep, \( N \), after \( t \) years will be given by the formula:

\[
N = \frac{330}{1 + 10 \cdot (0.79)^t}
\]

and that the sheep population will be able to maintain itself without further supervision once the population reaches a size of 85. (a) Graph \( N \) versus \( t \). (b) How many years must the state of Colorado maintain a program to care for the sheep? Show your work. (c) How many bighorn sheep can the environment in the protected area support? (hint: examine the graph for large values of \( t \).)

\[
(a) \quad N = \frac{330}{1 + 10 \cdot (0.79)^t} \quad \text{right hand side} \quad \frac{330}{1 + 10 \cdot 0.79^t}
\]
Alternatively (this is optional) we can do a little calculus and take the limit (left-sided limit) of the expression as $t$ goes to infinity.

\[ N = \frac{330}{1 + 10 \cdot (0.79)^t} \]

(b) Evaluate at point $t = 85$:
\[ 85 = \frac{330}{1 + 10 \cdot 0.79^{85}} = 5.277302835 \]

(c)
\[ N = \frac{330}{1 + 10 \cdot (0.79)^t} \]

Start of parameters -- change these for each version of the problem

Attachment: Version 2 of Sheep Script, with parameters

Version 2, with use of parameters
We call the size of the stable population \( s \).

\[
s := 85
\]

(a) Graph \( N \) versus \( t \). (b) How many years must the state of Colorado maintain a program to care for the sheep? Show your work. (c) How many bighorn sheep can the environment in the protected area support? (hint: examine the graph for large values of \( t \).)

(a) To make the graphing work all the time, we set the vertical axis to "P+30" rather than a fixed value. This will set the vertical axis so that it will be 30 more than the value of \( P \) which from the first problem we have realized is the top of the graph.

\[
sheepEquation := N = \frac{P}{1 + 10 \cdot (c)^t}
\]

\[
N = \frac{330}{1 + 10 \cdot 0.79^t}
\]
(b) Finding the time requires substituting the value of $s$ for $N$ in the equation, and then solving the resulting equation for $t$.

$$\textit{sheepEquation} \quad N = \frac{330}{1 + 10 \times 0.79^t} \quad 85 = \frac{330}{1 + 10 \times 0.79^t} \quad 5.277302835 \quad 5.277302835$$

Alternatively (this is optional) we can do a little calculus and take the limit (left-sided limit) of the expression as $t$ goes to infinity.

$$\textit{sheepEquation}$$

$$N = \frac{330}{1 + 10 \times 0.79^t}$$

$$\frac{330}{1 + 10 \times 0.79^t}$$

$$330.$$  

End of script

**Attachment: Version 3 of Sheep Script, with parameters**

**Version 3 with edited parameters and re-execution**

**Start of parameters -- change these for each version of the problem**

$$P := 450$$

$$450$$  

$$c := 0.83$$

$$0.83$$
We call the size of the stable population \( s \).

\[
s := 100
\]

We call the size of the stable population \( s \).

\[
N = \frac{450}{1 + 10^{0.83t}}
\]

End of parameters

(a) Graph \( N \) versus \( t \). (b) How many years must the state of Colorado maintain a program to care for the sheep? Show your work. (c) How many bighorn sheep can the environment in the protected area support? (hint: examine the graph for large values of \( t \).)

(a)

To make the graphing work all the time, we set the vertical axis to "P+30" rather than a fixed value. This will set the vertical axis so that it will be 30 more than the value of \( P \) which from the first problem we have realized is the top of the graph.

\[
sheepEquation
\]

\[
N = \frac{450}{1 + 10^{0.83t}}
\]

(4.45)

(4.47)
(b) Finding the time requires substituting the value of $s$ for $N$ in the equation, and then solving the resulting equation for $t$.

$$\text{sheepEquation} \quad N = \frac{450}{1 + 10.083^t} \quad 100 = \frac{450}{1 + 10.083^t} \quad 5.634221548 \quad 5.634221548$$

Alternatively (this is optional) we can do a little calculus and take the limit (left-sided limit) of the expression as $t$ goes to infinity.

$$\text{sheepEquation}$$

$$N = \frac{450}{1 + 10.083^t}$$  \hspace{1cm} (4.50)

$$\frac{450}{1 + 10.083^t}$$  \hspace{1cm} (4.51)

$$450.$$  \hspace{1cm} (4.52)

**End of script**

Note: a "bug" in Maple 13 causes duplicate graphs to sometimes appear during re-execution of the script. It's easy enough to delete the redundant copies.

### 4.10 Summary of Chapter 4 material

<table>
<thead>
<tr>
<th>Important word processing operations in Maple worksheet</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
</tr>
<tr>
<td>Switch entry to 2D Math mode</td>
</tr>
<tr>
<td>Switch entry to Text mode</td>
</tr>
<tr>
<td>Switch entry to Label mode</td>
</tr>
<tr>
<td>Evaluate</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General form</strong></td>
<td><strong>Examples</strong></td>
</tr>
</tbody>
</table>
| $symbol \ name := expression$ | $x := 5$

$$x = 5$$  \hspace{1cm} (4.53)

$$y := z + \frac{x^2}{2}$$

$$z + \frac{25}{2}$$  \hspace{1cm} (4.54)

$mySalary := (1,10,1) + 2.1e4$

$$21005.$$  \hspace{1cm} (4.55)

$myList := [a, b, x, y]$

$$[a, b, 5, z + \frac{25}{2}]$$  \hspace{1cm} (4.56)
<table>
<thead>
<tr>
<th>Type of structure</th>
<th>What they look like</th>
<th>Examples</th>
<th>Access (Select-&gt;i on right-click/control-click menu also works for access)</th>
<th>Empty structure value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic data structures in Maple</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequences</td>
<td>Values separated by a comma</td>
<td>19, 47, 92</td>
<td>(1.10.7)[1]</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>19, 47, 92 (4.59)</td>
<td>19 (4.60)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(1.10.7)[−1]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>92 (4.61)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(1.10.7)[1..2]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>19, 47 (4.62)</td>
<td></td>
</tr>
<tr>
<td>Lists</td>
<td>A sequence surrounded by square brackets [ ]</td>
<td>MyData := [1.0, x, \frac{3}{4}, a]</td>
<td>MyData[2]</td>
<td>[]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>x (4.64)</td>
<td>[] (4.66)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MyData[5]</td>
<td>Error, invalid subscript selector</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MyData[2..4]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>\left[\frac{3}{4}, a\right] (4.65)</td>
<td></td>
</tr>
<tr>
<td>Sets</td>
<td>A sequence surrounded by curly braces { }</td>
<td>Scores := {3, 7, 3, 10}</td>
<td>Scores[1]</td>
<td>{} (4.70)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{3, 7, 10} (4.67)</td>
<td>3 (4.68)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Scores[−3]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3 (4.69)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Scores[−4]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Error, invalid subscript selector</td>
<td></td>
</tr>
</tbody>
</table>

### Scripts

Creating a script | In a Maple worksheet, take a problem and solve it using the clickable interface or through textual operations.
Note similarities and differences between different versions of the problem and find parameters. Assign the parameters at the beginning of the script. Rework the rest of the script so that the formulas refer to the parameters by name, rather than the values used in the original version of the problem.

| Using a script | Copy and paste the script to a new location  
|                | Edit the assignments to reflect the new version of the problem.  
|                | Edit->Execute->Selection, or just hit enter (return on Macintosh) multiple times to perform the operations in the new version of the script. |

| Rationale for using scripts | More work to do than clickable interface the first time. Saves time if you expect to want to reuse the operations on multiple versions of the same problem. |
# Chapter 5 More sophisticated scripting

## 5.1 Chapter Overview

We introduce textual entry of `solve` and `plot` operations. Textual entry is often preferred by programmers because it is easier to edit scripts written through textual means.

To exploit the greater expressiveness of textual programming, the use of lists, sets, and sequences in `solve` and `plot` is explained and illustrated.

Programmers rely on the on-line documentation to manage the complexity of remembering the details of a knowledge-intensive system such as Maple. They learn/remember how to use a feature by looking up the description, finding an example close to what is desired, and then actively experiment with the example in a fresh worksheet. Reading without experimentation is not very productive.

Evaluation is defined -- it is what causes Maple to perform a calculation and possibly assign the result to a label or a variable. You cause evaluation by hitting `enter` after typing in an expression in a dotted rectangle, or by doing `Edit->Execute...`

If you open a previously saved worksheet, you have to (re)execute it to make its appearance and state agree again.

## 5.2 Textual entry of directions compared to palette selection

The clickable interface is a good way to get a calculation done quickly, but the actions specified in this way are hard to edit when building scripts. Maple, like most languages, has a textual version of all operations it performs. The editing involved in scripting development is easier to do on the textual version. Furthermore, the number and variety of operations available in textual form is far greater than what's available in the clickable interface, giving the textual programmer much greater power and flexibility. The downside of using the textual version is that the developer must enter the entire text correctly, rather than the fill-in-the-blank philosophy of clicking. Fortunately, the interactive interface allows one to edit failed attempts and retry, so "perfect entry" is not necessary.

Part of the transition that technical users make in transitioning from reusing someone else's work to creating new works of their own comes in becoming proficient with the use and creation of textual forms. Without such proficiency, it is hard to realize the full power of the computer in modeling and simulation situations.

The textual form of an operation in Maple has the general form:

\[
\text{operationName} \left( \text{sequence of values} \right)
\]

The `operationName` can be something like `solve` or `plot`. If the `sequence of values` has more than one value in it, then each value is separated by a comma. The technical term for the "values" in this situation is actual parameter or argument. Note that the parentheses around the sequence of values are mandatory -- you will either get an error or a result that's far from what you want if you omit the parentheses.

### General format of textual operations in Maple, with examples and mistakes

<table>
<thead>
<tr>
<th><code>operationName</code> (sequence of arguments)</th>
<th>Sometimes the term actual parameter is used as a synonym for argument.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Examples</td>
<td></td>
</tr>
<tr>
<td>1.</td>
<td></td>
</tr>
<tr>
<td><code>solve(x=3 \cdot x^2 - 2, x)</code></td>
<td>Here the operation name is <code>solve</code>. There are two arguments, the equation ( x=3 \cdot x^2 - 2 ) and the symbol ( x ).</td>
</tr>
<tr>
<td>( x = \frac{2}{3}, 1 )</td>
<td>Note that the form of the answer returned by the textual version is slightly different from invoking <code>solve</code> through the clickable interface. The former is actually a version easier to deal with in scripts.</td>
</tr>
</tbody>
</table>

55
What you get if you leave out the parentheses -- not much!

You get another one of those "unable to match delimiters" messages if you forget one of the parentheses.

This is what happens if we forget to use the right arrow key to descend from the exponent of \( x^2 \). Maple thinks that we are talking about \( x \) to a power called 2-2,\( x \), which understandably it doesn't think is "valid".

If the second argument is just the variable, then \( \text{plot} \) uses default values for the range.

No plotting happens if we forget the mandatory parentheses.

This can be mystifying if you thought you were trying to specify the horizontal range. There is no error message, but the picture is not right. If you look at the on-line documentation for help, you will see that giving \( \text{plot} \) three arguments means something different -- the third argument can be taken as the value of the vertical range of a plot.

The only way to fix this is to know that this picture, while not generating any error message, is not what is wanted. That will spur a search for an inaccuracy in the textual form and its eventual correction.
5.3 Lists and plotting. Strings

Recall that lists in Maple are (page 41). You specify a list by listing the items in the list, enclosed in square brackets [].

If the first argument to *plot* is a list of expressions, then *plot* will on a single graph display the plots of all the expressions in the set. Each one will be displayed in a different color.

**Plotting of multiple expressions**

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
</table>
| `plot([x^2, sin(x^2)], x = 0 .. 4)` | Plot two expressions on a range where both have comparable-sized results. Here we use the list `[x^2, sin(x^2)]` to indicate the two expressions that should be plotted. Problem: approximately where is the expression 
\[-3 \cdot x^2 + x - 2\] equal to 5? While we could use solve to tell us exactly, it's often worthwhile to draw a picture and process the situation visually. If we give plot the two expressions 
\[-5\] and \[-3 \cdot x^2 + x - 2\] then plot will plot not only the parabola (the second expression, it will also plot the expression that is always -5 for any value of x. This corresponds to the horizontal line drawn on the plot. Visually we can see that the parabola is -5 at roughly -0.8 and 1.2. We can even get a little more precise by If we click on the plot and then position the mouse over the plot area (page 19). |

`plot([-5, x - 3 \cdot x^2 - 2], x = -3 .. 3)`
Plotting multiple expressions simultaneously can be useful when you want to compare them. Assuming that the scales are comparable, one can get a sense of similarity or dissimilarity "at a glance".

Plotting can be used with the textual version of `plot` . If we give the textual version of the `plot` operation two lists of numbers that have the same length, then `plot` will regard the first list as a list of x-coordinates, and the second list as corresponding y-coordinates. If you provide `plot` with the third argument `style=point`, then it will produce a point plot.

### Plotting points with lists of numbers

<table>
<thead>
<tr>
<th>xList := [1, 2, 3, 4]</th>
<th>This plots the points (1,5), (2,6), (3,7) and (4,-1).</th>
</tr>
</thead>
<tbody>
<tr>
<td>yList := [5, 6, 7, -1]</td>
<td>Without the third argument, <code>plot</code> will try to connect the points with a curve.</td>
</tr>
<tr>
<td></td>
<td>Maple cares about whether things are capitalized or not. <code>Style</code> is not the same as <code>style</code>.</td>
</tr>
</tbody>
</table>

```plaintext
plot(xList, yList, style=point)
```

```plaintext
plot(xList, yList)
```
A string in Maple is something enclosed in double-quotes: "red", "this is a string?", "Bl****182+++" are all strings. The double-quote symbol is mandatory for a string. Single-quotes ' (also known as apostrophes), backquotes ` (also known as acute accent marks) are not substitutes for double-quotes in writing Maple strings. Characters enclosed by apostrophes or backquotes mean something else to Maple. Indiscriminate use of punctuation marks will lead to undesired results if not error messages.

In addition to being used to input data points in plot, lists also can be used in specifying colors and axis labels.

**Plot options: colors and labels**

If one of the arguments to the plot operation is of the form
\[
\text{color = list of color names}
\]
then those colors will be used. Most reasonable names will work, but the full list can be seen. Note that we are using the textual version of the symbolic constant π. Many people, find it easier to do than selecting the symbol with the Common Symbols palette since it becomes easy to remember because it occurs frequently.

Note that because the expressions being plotted are given in a set, the color assignment is not in the same order that they were typed in. If we wanted the same order, we should give the plot expressions in a list.

Note that giving a set of colors doesn't work. Neither does forgetting to brackets for the list of colors.

One of the proficiency issues with textual input is that you have to remember all the ( ) parentheses and [ ] brackets. Can you find the the missing delimiter(s)? If you leave enough delimiters out, you get error messages that don't complain about missing delimiters. You have to infer that there's something wrong around the point where you mentioned "sin". Computers aren't very good at figuring out what you meant to type when there are so many possibilities.
5.4 Learning through on-line documentation and experimentation

All the options available in the Plot Builder available through the right-click (control-click) interface are also available in the textual version of `plot`. In fact, there are many additional options and varieties of plotting available. The way to find out what the features are and how to invoke them is to consult the on-line documentation.

We can find out more about the textual forms of plotting by invoking Help -> Maple Help and typing `plot` into the search field. When we do so, we see the information in the figure below:

If one of the arguments to the `plot` operation is of the form `labels = list of aestitiles` then those will be used.
We scroll to the bottom of the page and find an example of this. We are looking for a version of plots where v1 and v2 are lists. We don't see something exactly like that but we do see something with Vectors which are similar. Since the document says this should work for lists or vectors, we take the example and see if we can modify it for our own purposes:
Examples of plot

Evidently, the first list is the values of the $x$ (horizontal) coordinates, and the second list the values of the $y$ (vertical) coordinate.

We copy and paste the example into a Maple worksheet and then see if we can get it to work.

Copying the example

According to the documentation, we should be able to get this to work if the first two arguments are lists or vectors. So we edit the example to do lists instead and re-execute the line to see if it works in the same way.
Modifying the example

To learn about plot options such as colors and labels, we click on the plot,options item under the search results for plot (see green oval in the figure). Clicking on that item produces this information. We see information about color (with another link to see colors), along with possibilities, for labels, symbols, styles, etc. Again, the way to learn the options is through copying and pasting the examples into a fresh worksheet, getting them to work, and then modifying them to suit your own purposes.

Plot option help page
Note the "colour". Like Blackberries which are designed just down the road from the Maple company, this is a Canadian product. You will see things like this as well as other indications that it's not an all-American world out there. For example, you can convert pints into Imperial Gallons through Tools -> Assistants -> Units Calculator.

### 5.5 Operations on lists

So far we have talked only about creating lists, and assigning lists as the value of a variable. You can also generate a sublist of a list, generate a particular item in a list, count the number of items in the list, and convert a list into other types of data.

#### Operations on lists

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a list</td>
<td>( s1 := [a, b, c, a] )</td>
<td>Lists can contain symbols, numbers, expressions -- anything, even other lists.</td>
</tr>
<tr>
<td>Specify a sublist of values</td>
<td>( s1 := [a, b, c, a] )</td>
<td>If a list is followed by another pair of braces with a range inside, then a sublist is computed as a result. Here we have the list that's the first through third items of ( s1 ).</td>
</tr>
<tr>
<td>Specify one item from the list</td>
<td>( s2 := [1, 3.47, 97, -5.9, 2.1] )</td>
<td></td>
</tr>
<tr>
<td>Specify a sublist with one item</td>
<td>( [1, 3.47, 97, -5.9, 2.1] )</td>
<td></td>
</tr>
<tr>
<td>Count the number of items in the list</td>
<td>( s1 := s1[1..3] )</td>
<td></td>
</tr>
<tr>
<td>Add together all the items in the list</td>
<td>( [a, b, c] )</td>
<td></td>
</tr>
<tr>
<td>Compute the average of all the numbers in the list</td>
<td>( \frac{\sum_{i=1}^n s2[i]}{n} )</td>
<td></td>
</tr>
<tr>
<td>Convert a list into a sequence</td>
<td>( op(ts1) )</td>
<td>op has an idiosyncratic name for an operation. It stands for &quot;operands&quot;.</td>
</tr>
<tr>
<td>Convert a list into a set</td>
<td>( a, b, c )</td>
<td>Since sets contain no repetitions, the second &quot;a&quot; doesn't appear in the set.</td>
</tr>
<tr>
<td>Convert a list into a string</td>
<td>( convert(s1, set) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( {a, b, c} )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( convert(s2, string) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;[1, 3.47, 97, -5.9, 2.1]&quot;</td>
<td></td>
</tr>
</tbody>
</table>
5.6 solve, lists and sequences

To solve a system of equations, use a list of expressions or equations for the first argument to `solve`. Use a list of variables as the second argument. `solve` will return a sequence of lists as the result.

**Solving systems of equations with solve**

<table>
<thead>
<tr>
<th>eqn := 3*x = x^2 - 28</th>
<th>We get a sequence of solutions since there is a double root.</th>
</tr>
</thead>
<tbody>
<tr>
<td>solve(eqn,x) == -4.7</td>
<td>We evaluate the equation at the first root and see that it does satisfy the equation.</td>
</tr>
<tr>
<td>eval(eqn,x=solns[1]) == -12 = -12</td>
<td>Same for the second.</td>
</tr>
<tr>
<td>eval(eqn,x=solns[2]) == 21 = 21</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>system := [3<em>x + 5</em>y = 6, 2*x - 5 = y]</th>
<th>We want to assign a set as the value of the variable system. Maple tells us that the name is already in use as a built-in function, so it won't let us do that.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error, attempting to assign to 'system' which is protected</td>
<td></td>
</tr>
<tr>
<td>sys := [3<em>x + 5</em>y = 6, 2*x - 5 = y]</td>
<td>We choose a different variable to assign the set to.</td>
</tr>
<tr>
<td>vars := [x,y]</td>
<td>We specify the set of variables we want to solve for, then call <code>solve</code>. We get a list with one element (which itself is a list) as a solution.</td>
</tr>
<tr>
<td>solns := solve(sys,vars)</td>
<td>We extract the first element of the list. Notice that that are fewer [ ]s.</td>
</tr>
<tr>
<td>solns[1] = [x = 31/13, y = -3/13]</td>
<td>We evaluate the system at the solution that <code>solve</code> has found and verify that this really does satisfy the system of equations.</td>
</tr>
<tr>
<td>eval(sys,solns[1]) == 6 = 6, -3/13 = -3/13</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sys2 := [x^2 + y^2 = 25, x + y = 5]</th>
<th>This system of equations has two distinct solutions, so we get a list with two elements in it. Each element is a distinct solution.</th>
</tr>
</thead>
<tbody>
<tr>
<td>[x^2 + y^2 = 25, x + y = 5]</td>
<td></td>
</tr>
<tr>
<td>solve(sys2,[x,y])</td>
<td></td>
</tr>
<tr>
<td>[[x = 5,y = 0], [x = 0,y = 5]]</td>
<td></td>
</tr>
</tbody>
</table>
5.7 What is evaluation?

We've informally talked about "evaluation". Evaluation occurs when you type something into Maple and then hit the enter key (or control-=). The computer acts on what you typed. It looks to see if it knows any calculational rules associated with the input. If it does, it will perform the rules as they are recorded within its programming.

For example, if we type \( \frac{2}{3} + \frac{5}{7} \) and then hit control-=, we would see something like this: \( \frac{2}{3} + \frac{5}{7} = \frac{29}{21} \).

But let's go over this stroke by stroke:

16. As we type in the numbers and the arithmetic operators, Maple is listening to each keystroke. It tries to display what it has according to the rules of mathematical writing. But it doesn't perform any arithmetic yet because we haven't asked it to evaluate the expression we've entered (because it really can't make sense until we've finished entering it).

17. When we hit enter, Maple starts to evaluate the expression -- to perform the arithmetic, remove greatest common divisors, and so forth.

18. When the calculation is complete, Maple displays the result in 2 dimensional math notation, and gives it a label that we can use later on.

Maple is known as an interactive, interpreted system because of its behavior of evaluating a line of input as soon as we hit enter. Other languages, such as Java or C++, require you to enter many lines of notation before doing any calculations. This approach has its advantages, particularly when you want to do more complicated calculations that need many steps before they come to fruition. But interpreted systems have their place as environments where you can experiment and develop modest things quickly.

5.8 Worksheet appearance is not the same as worksheet state

When you are running Maple, there are two things that it keeps track of: the appearance of the worksheet: the characters, formulas, pictures, plots, etc. that are displayed and the state: the values of the variables that have been assigned since execution began. The two are not the same.

When you open a worksheet that has been previously saved, you may see a lot of instructions, words, and results. However, the state is (almost) empty. No values have been assigned to variables (regardless of appearance) because nothing has been evaluated. Maple does not do the instructions listed in the worksheet until evaluation occurs -- which will occur only if you hit enter on a particular instruction, perform Edit->Execute->Worksheet, etc.

Thus, this rule of thumb: after you open a worksheet, execute the portion of it that you wish the state to reflect.

5.9 Attachment: Textual versions of terms from the Expression, Common Symbols, and Greek Palettes

<table>
<thead>
<tr>
<th>Operations from the palette</th>
<th>Textual form</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a + b ), ( a - b ), ( a \cdot b ), ( a^b ), ( \sqrt{a} ), ( a^{\frac{1}{n}} ), ( a! ), ( e^a ), ( \log_{10}(a) ), ( \log_{b}(a) ) ( \log_{a}(a) ) ( \log_{a}(a) ) ( \log_{b}(a) ) ( \log_{a}(a) )</td>
<td>( a+b ), ( a-b ), ( a*b ), ( a^b ), ( a[n] ), ( \sqrt{a} ), ( a^{1/n} ), ( abs(a) ), ( \exp(a) ), ( \log<a href="a">10</a> ), ( \log<a href="a">b</a> )</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\binom{\frac{5}{3}}{\frac{1}{3}, f(x)}
\end{align*}
\]
### 5.10 Summary of Chapter 5 material

**Troubleshooting textual input in Maple**

<table>
<thead>
<tr>
<th>Remember to...</th>
<th>Examples with error(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supply a function name and arguments (parameters)</td>
<td><code>solve( )</code></td>
</tr>
<tr>
<td>Match delimiters (parenthesis and brackets)</td>
<td><code>solve(x = 3 \cdot x^2 - 2, x)</code></td>
</tr>
<tr>
<td>Press the right arrow key to exit from variable exponents</td>
<td><code>solve(x = 3 \cdot x^2 - 2, x)</code></td>
</tr>
<tr>
<td>Set ranges correctly when plotting</td>
<td><code>plot(x = 3 \cdot x^2 - 2, x = -3 \cdot 3)</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Plotting</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plotting multiple expressions</td>
<td><code>plot([x^2, \sin(x^2)], x = 0..4)</code></td>
</tr>
</tbody>
</table>
| Plotting with lists | `xList := [1, 2, 3, 4]
[1, 2, 3, 4]` (5.43)
`yList := [5, 6, 7,-1]
[5, 6, 7,-1]` (5.44)
`plot(xList, yList, style = point)` |
| Using multiple colors in a multi-plot | `plot([\sin(x), \sin(x/2), \sin(2-x)], x = -4..Pi..Pi, color = ["red", "green", "blue"])` |
| Set the titles of the axes | `plot(3.5 \cdot x^2 - 2, x = 1..5, labels = ["temperature (in degrees C)", "pressure in kilopascals"])` |

**Using Maple's built-in help**

Use Help>Maple Help or press Ctrl-F1 (Command-F1 on a Mac) Remember that you can click on related topics when viewing the help for a particular command

<table>
<thead>
<tr>
<th>Operations on lists</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a list</td>
<td><code>sl := [a, b, c, a]</code></td>
</tr>
<tr>
<td>Specify a sublist of values</td>
<td><code>[a, b, c, a]</code> (5.45)</td>
</tr>
<tr>
<td>Specification</td>
<td>Example</td>
</tr>
<tr>
<td>---------------</td>
<td>---------</td>
</tr>
<tr>
<td>Specify one item from the list</td>
<td>$tsl := sl[1..3]$</td>
</tr>
<tr>
<td>Specify a sublist with one item</td>
<td>$[a, b, c]$ (5.46)</td>
</tr>
<tr>
<td>Count the number of items in the list</td>
<td>$sl[1]$</td>
</tr>
<tr>
<td>Add together all the items in the list</td>
<td>$a$ (5.47)</td>
</tr>
<tr>
<td>Compute the average of all the numbers in the list.</td>
<td>$s2[3..3]$</td>
</tr>
<tr>
<td></td>
<td>$[97]$ (5.48)</td>
</tr>
<tr>
<td></td>
<td>$n := nops(s2)$</td>
</tr>
<tr>
<td></td>
<td>$5$ (5.49)</td>
</tr>
<tr>
<td></td>
<td>$\sum_{i=1}^{n} s2[i]$</td>
</tr>
<tr>
<td></td>
<td>$97.67$ (5.50)</td>
</tr>
<tr>
<td></td>
<td>$\frac{\sum_{i=1}^{n} s2[i]}{n}$</td>
</tr>
<tr>
<td></td>
<td>$19.53400000$ (5.51)</td>
</tr>
<tr>
<td>Convert a list into a sequence</td>
<td>$op(tsl)$</td>
</tr>
<tr>
<td>Convert a list into a set</td>
<td>$a, b, c$ (5.52)</td>
</tr>
<tr>
<td>Convert a list into a string</td>
<td>$convert(sl, set)$</td>
</tr>
<tr>
<td></td>
<td>${a, b, c}$ (5.53)</td>
</tr>
<tr>
<td></td>
<td>$convert(s2, string)$</td>
</tr>
<tr>
<td></td>
<td>$'[1, 3.47, 97, -5.9, 2.1]'$ (5.54)</td>
</tr>
<tr>
<td>Solving a system of equations using lists</td>
<td>Examples</td>
</tr>
<tr>
<td>Create the system of equations</td>
<td>$sys := [3x + 5y = 6, 2x - 5 = y]$</td>
</tr>
<tr>
<td></td>
<td>$[3x + 5y = 6, 2x - 5 = y]$ (5.55)</td>
</tr>
<tr>
<td>Set the variables of the system</td>
<td>$vars := [x, y]$</td>
</tr>
<tr>
<td></td>
<td>$[x, y]$ (5.56)</td>
</tr>
<tr>
<td>Solve the system and extract the first solution of possibly many solutions</td>
<td>$solns := solve(sys, vars)[1]$</td>
</tr>
<tr>
<td></td>
<td>$\left[ x = \frac{31}{13}, y = -\frac{3}{13} \right]$ (5.57)</td>
</tr>
</tbody>
</table>
6 Chapter 6 Using and Defining Functions

6.1 Chapter overview

Functions occur so much in mathematics that it's natural that Maple knows a lot about them and how to compute with them.

You can also define your own functions.

There are a few pitfalls in the use of functions in Maple:

a) The names of common mathematical functions used in Maple may differ from what you are used to.

b) Some functions in Maple do non-mathematical things, such as solve, and plot. Others take novel arguments -- lists, sets, equations, and ranges, rather than numbers.

c) The way functions are defined uses := and -> rather than the use of = as found in math textbooks. This is because the "context-free" language processing of Maple thinks an equation is being defined whenever it sees an equal sign.

6.2 Functions in computer languages: a way of producing an output from inputs

Everyone is introduced to the idea of a function in secondary school mathematics: Calculators can compute many of the common functions found in high school algebra and pre-calculus: \( \sin, \cos, \ln, \sqrt{} \), etc.

Maple can evaluate these functions. The common ones are found in the Expression palette but there are hundreds more.

When entering a functional expression, the syntax used is:

\[
\text{function name ( sequence of arguments )}
\]

The parentheses are **mandatory** in Maple. Unexpected results, possibly including an error message may result if you forget them.

**Function results in Maple**

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sin(\pi) )</td>
<td>All the common functions know how to give limited-precision results given limited precision inputs</td>
</tr>
<tr>
<td>( \cos \left( \frac{\pi}{2} \right) )</td>
<td>Maple's programming will return an exact number if the function has that kind of result for the given input. However, if a simple exact result can't be found, Maple will return what you typed in. Sometimes there is a little simplification that goes on so what comes out is not literally what comes in, although it usually obvious that it is mathematically equivalent. For example, Maple will always simplify fractions by eliminating the greatest common divisor from the numerator and denominator.</td>
</tr>
<tr>
<td>( \cos(a, b, c) )</td>
<td>This doesn't work because cosine expect an algebraic expression, not a list.</td>
</tr>
</tbody>
</table>
| \( \cos(2 \pi) \) | This mistake might come about if you typed a comma instead of a *.
| \( \cos(\pi) \) | |

\( 6.1 \)
### 6.3 How can I remember so many functions?

A well-developed system such as Maple, Matlab, or C# has thousands of built-in functions. It is also unreasonable to expect that you can get by by knowing only three or four functions. We've already listed a few dozen (some math functions, such as sin, some operations such as factor). So the bad news is that you will have to remember at least the names of a several dozen functions. The good news is that learning about functions is not that taxing -- if you own a scientific calculator you've already dealt with a situation where you can operate a dozen functions.

A reasonable stance to take is to be familiar (i.e. know "by heart") functions you use often and to be adept at using documentation to look up the details of the ones that you need only occasionally. In exams and test about computer functions, you may be quizzed on the details of the most common, but things never degenerate into a situation where your grade will depend on how well you memorize phone book-sized lists.

A quick-recall method for access to common functions with textual entry is to type the escape key (Esc) after typing a few characters. A pop-up menu will appear that will list possible ways of completing what you typed. If what you are entering is a Maple operation such as `solve`, this will provide a textual template that may be easier to fill in than to recall all the details of. This is called command completion.

**Command Completion**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>solve</code></td>
<td>We type <code>solve</code> and then the ESC key. A pop up menu shows option, including several forms of <code>solve</code>.</td>
</tr>
</tbody>
</table>
Clicking on one of the options provides something to fill out. We are not compelled to have a second variable of \( x \), it's just shorthand reminder that the second argument is the variable and the first one is the equation.

Experienced users refer to the on-line documentation to help remember details about functions. As has mentioned earlier, this is available through Help -> Maple Help menu feature (key shortcut: press the control key and then the F1 key). If you recall a phrase or a name of a function, you can type it into the search field and the on-line help system will, like Google, produce the pages it has about your text entry. You can then explore further by pressing links. Trying the examples typically given at the end of the description is a good way to get a form that you can use for your own purposes.

Using on-line help

We want to find information about how to use the inverse sine function in Maple. We start up on-line help and type in "inverse trigonometric" into the search field, then hit the "search" button. The page we see does tell us that it's probably called "arcsin" but we'd like to see more. We see a link in the "see also" which we click on.

This uncovers more links. The one that says "invtrig" seems promising so we click on that link.
6.4 Defining your own functions with -> (arrow)

Maple allows you to define simple functions with the use of ->. The general form is

function name := (sequence of arguments) -> expression that describes result.

These can be entered through the Expression Palette, or textually. The arrow is entered textually by typing a - and then a >, with no spaces separating them. We give an example of function definition in the following example.

A function definition in a math textbook, and a problem associated with it (from Precalculus)

Fruit flies are placed in a half-pint milk bottle with a banana (for food) and yeast plants (for food and to provide a stimulus to lay eggs). Suppose that the fruit fly population after \( t \) days is given by

\[
P(t) = \frac{230}{1 + 56.5 \cdot e^{-0.37t}}
\]

(a) What is the population after five days?

(b) How long does it take for the population to reach 180?

Analyzing the text, we see that it defines a function named \( P \). It takes one input (argument), \( t \), and produces as output whatever you get from evaluating the expression \(
\frac{230}{1 + 56.5 \cdot e^{-0.37t}}
\). Assuming that Maple understands the definition of \( P \) like it does a built-in function, then the description of what happens when you evaluate \( P(5) \) would be: "Substitute 5 for wherever you see \( t \) in the expression \(
\frac{230}{1 + 56.5 \cdot e^{-0.37 \cdot 5}}
\). This gives you

\[
\frac{230}{1 + 56.5 \cdot e^{-0.37 \cdot 5}}
\]

. Perform all the arithmetic and relevant simplifications in this expression and return that as the result of the function."

Even though 5 is an exact number, because there are limited precision numbers in the expression we expect that the result will be a limited-precision number. If the expression had only exact numbers in it, then the calculation would be done exactly.

What we would like to do is to tell Maple about the definition of \( P \) and use it in our work.
We can do this through the clickable interface. We anticipate reuse of this for other days and population levels, and turn it into a parameterized script:

**User-defined functions through the Expression Palette**

<table>
<thead>
<tr>
<th>Start of parameters</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>numDays := 5</td>
<td>The general form is from the ( f = a \rightarrow y ) in the Expression palette. We alter slots in the template to mention the specific function by name, the name of the input, and the expression that describes how to calculate the output.</td>
</tr>
<tr>
<td>popLevel := 180</td>
<td>With this definition, ( P(5) ) works exactly as it should.</td>
</tr>
</tbody>
</table>

End of parameters

\[
P(t) = \frac{230}{1 + 56.5e^{-0.37t}}
\]

Compute when the population reaches desired level by solving the equation \( P(t) = \text{popLevel} \).

\[
P(t) = \text{popLevel} \\quad \text{solve} \quad \{ t = 14.36533644, t = 14.36533644, t = 14.36533644 \}
\]

We plot the function to see how the population grows. This is not needed by the problem but it helps us understand the situation better.

We compute the number flies after \( \text{numDays} = 5 \) days.

\[
P(\text{numDays})
\]

\[
23.27016688
\]

End of script
***User-defined functions through the Expression Palette***

Note that in order to explain what we were doing with the plot, we gave the textual form rather than the clickable form.

\[
\text{function name} := \text{input name} \rightarrow \text{expression involving constants and input}
\]

**User-defined function textually**

<table>
<thead>
<tr>
<th>Function definition</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Start of parameters</strong></td>
<td></td>
</tr>
<tr>
<td>( \text{numDays} := 5 )</td>
<td></td>
</tr>
<tr>
<td>( \text{popLevel} := 180 )</td>
<td></td>
</tr>
<tr>
<td><strong>End of parameters</strong></td>
<td></td>
</tr>
<tr>
<td>( P := t \rightarrow \frac{230}{1 + 56.5 e^{-0.37 t}} )</td>
<td>Even though we entered the function definition in a different way, the same thing happened -- a function was defined and ( P ) is its name. To get the right arrow, we typed ( \rightarrow ), which Maple re-typeset as a right arrow.</td>
</tr>
<tr>
<td>( t \rightarrow \frac{230}{1 + 56.5 e^{-1.070 t}} )</td>
<td>About the only other difference is the use of the textual version of solve. Note that the form of the answer is slightly different (actually, easier to use).</td>
</tr>
</tbody>
</table>

Compute the number flies after \( \text{numDays} = 5 \) days.

\[
P(\text{numDays})
\]

\[
23.27016688
\]

We plot the function to see how the population grows. This is not needed by the problem but it helps us understand the situation better.

```maple
plot(P(t), t = 0 .. \text{numDays}, title = "Well maybe they don't like bananas", labels = ["t", "# of flies"])
```

Compute when the population reaches desired level by solving the equation \( P(t) = \text{popLevel} \).

```maple
soln := solve(P(t) = \text{popLevel}, t)
```
6.5 Troubleshooting function definitions

Function definition is another place where the standard notation in mathematics does not work in Maple. Recall that the technology that is standard in most computer language understanding systems needs to assign a unique meaning to input from the way it looks. Equations already use "==" so if you use f(x) = ... Maple will understand you to be talking about an equation, not a function definition. Use :=

Troubleshooting function definitions

<table>
<thead>
<tr>
<th>Function definition</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F := (m_1, m_2, r) \rightarrow \frac{g \cdot m_1 \cdot m_2}{r^2} )</td>
<td>This defines a function for the gravitational attraction force. ( m_1, m_2 ) and ( r ) are the arguments. ( g ) is another symbol used in the expression but it isn’t included as an argument.</td>
</tr>
<tr>
<td>( g := 6.673 \times 10^{-11} )</td>
<td>We define the value of ( g ). We expect not to change this while using the function. We found the value by typing &quot;gravitation&quot; into Maple on-line help.</td>
</tr>
<tr>
<td>( F(1, 2, 10) )</td>
<td>This calculates the attraction force between a 1 kilogram and 2 kilogram mass that is 10 meters apart. The force is in units of Newtons.</td>
</tr>
<tr>
<td>( F_{bad} := (m_1, m_2, r) \rightarrow \frac{g \cdot m_1 \cdot m_2}{r^2} )</td>
<td>This doesn’t work at all. Do you see the difference between the definition of ( F ) and ( F_{bad} )? One uses assignment := the other is either mistyped or mistaken.</td>
</tr>
<tr>
<td>( F := (m_1, m_2, r) \rightarrow \frac{g \cdot m_1 \cdot m_2}{r^2} )</td>
<td>This tries to use &quot;=&quot; instead of &quot;-&gt;&quot; to define the function ( F ). There is no error message, but we get an unexpected expression rather than the number we were expecting. Someone could use this gibberish in further work if they weren’t able to discern that the result isn’t numerical as they were expecting.</td>
</tr>
<tr>
<td>( F_{bad}(1, 2, 10) )</td>
<td></td>
</tr>
<tr>
<td>( (m_1(1, 2, 10), m_2(1, 2, 10), r(1, 2, 10)) )</td>
<td></td>
</tr>
<tr>
<td>( F(1, 2, 10) )</td>
<td></td>
</tr>
<tr>
<td>( F2(m_1, m_2, r) := \frac{g \cdot m_1 \cdot m_2}{r^2} )</td>
<td>This actually works although it is not the form prescribed by these notes. Rather than &quot;=&quot; as would appear in a math textbook, the assignment operation &quot;:=&quot; is used instead. This produces the following pop-up:</td>
</tr>
</tbody>
</table>
6.6 The advantages of using functions in scripts

The advantage of using functions is *reuse does not require typing in the rule again*. With built-in functions, someone else did the programming of how to compute sin, cos, ln, etc., so you are just reusing their work. When you define your own functions, you expect to use them enough to save work after a few uses, even though it's more work to create the function in the first place.

Another benefit from defining functions is by giving them a name, a reader can more easily understand your documented work. It's a lot to talk about \( P \), the function that describes population, rather than \( \frac{230}{1 + 56.5 \cdot e^{-0.37 \cdot t}} \) all the time.

6.7 Attachment: some built-in problem-solving functions

The functions in the Expression palette have the same name and work similarly to those described in math textbooks. The operations discussed in this attachment are also found in math textbooks, but they are usually not given function names. It may seem novel to you that the rules for solving equations, factoring polynomials, or plotting can be collected together and given a function name. Yet this way of writing about such actions allows us to combine mathematics and working on it. Thus `solve, plot, factor`, etc. are true functions -- they have names, they are invoked with arguments, and return results that can be assigned to a variable.

<table>
<thead>
<tr>
<th>Textual names of common operations in Maple</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>Textual name of function</td>
</tr>
<tr>
<td>solve an expression or an equation</td>
<td>solve</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>solve an expression or an equation numerically</td>
<td>fsolve</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>factor an expression</td>
<td>factor</td>
</tr>
<tr>
<td>plot an expression</td>
<td>plot</td>
</tr>
<tr>
<td>evaluate at a point</td>
<td>eval</td>
</tr>
<tr>
<td>rhs ( x = 3x^2 + 1 )</td>
<td>( 3x^2 + 1 ) \hspace{1cm} (6.35)</td>
</tr>
<tr>
<td>( \text{rhs}(x = 0..10) )</td>
<td>( 0..10 ) \hspace{1cm} (6.37)</td>
</tr>
<tr>
<td>right hand side, left hand side of an equation</td>
<td>lhs ( x = 3x^2 + 1 )</td>
</tr>
</tbody>
</table>
| }
approximate with a limited-precision number

$$\text{evalf}$$

$$\text{evalf}(\pi)$$

$$3.141592654$$ \hspace{1cm} (6.38)

$$\text{evalf}(\pi, 20)$$

$$3.1415926535897932385$$ \hspace{1cm} (6.39)

$$\text{evalf}\left(\sin\left(\frac{\pi}{10}\right), 15\right)$$

$$0.309016994374947$$ \hspace{1cm} (6.40)

evalf has an optional second argument. If it's not there, Maple will compute a 10 digit approximation. If the second argument provided is a positive integer, then Maple will compute that many digits.

### 6.8 Summary of Chapter 6 material

<table>
<thead>
<tr>
<th>Defining custom functions with the arrow ((\rightarrow)) notation</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a custom function by naming it and its parameters. A custom function defined using arrow notation is of the form:</td>
<td>$\rho \leftarrow \frac{230}{1 + 56.5 \cdot e^{-0.37 \cdot t}}$</td>
</tr>
<tr>
<td>FunctionName := ParameterName (\rightarrow) Function</td>
<td>$t \leftarrow \frac{230}{1 + 56.5 \cdot e^{-0.37 \cdot t}}$ \hspace{1cm} (6.41)</td>
</tr>
<tr>
<td>In this case the function name is $\rho$ and its sole parameter is $t$. For multiple parameters, use the form:</td>
<td>$c \leftarrow (a, b) \leftarrow \sqrt{a^2 + b^2}$</td>
</tr>
<tr>
<td>FunctionName := (ParameterName1, ParameterName2, ..) (\rightarrow) Function</td>
<td>$(a, b) \leftarrow \sqrt{a^2 + b^2}$ \hspace{1cm} (6.42)</td>
</tr>
<tr>
<td>To call the function, use the form:</td>
<td>$numDays \leftarrow 5$</td>
</tr>
<tr>
<td>FunctionName(ParameterValues)</td>
<td>$5$ \hspace{1cm} (6.43)</td>
</tr>
<tr>
<td>$P(numDays)$</td>
<td>$23.27016688$ \hspace{1cm} (6.44)</td>
</tr>
<tr>
<td>To plot the function, set the range of an independent variable, and use it as a parameter to your custom function.</td>
<td>plot($P(t), t = 0 .. numDays$)</td>
</tr>
<tr>
<td>To solve the function for a particular value, use the solve function.</td>
<td>$popLevel \leftarrow 180$</td>
</tr>
<tr>
<td>$solve(P(t) = popLevel, t)$</td>
<td>$180$ \hspace{1cm} (6.45)</td>
</tr>
<tr>
<td>$soln := solve(P(t) = popLevel, t)$</td>
<td>$14.36533644$ \hspace{1cm} (6.46)</td>
</tr>
</tbody>
</table>

### Troubleshooting function definitions

<table>
<thead>
<tr>
<th>Error</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forgetting to use the assignment ((\leftarrow)) operator.</td>
<td>$F_{ba} = (m_1, m_2, r) \leftarrow \frac{g \cdot m_1 \cdot m_2}{r^2}$</td>
</tr>
<tr>
<td>Error, invalid operator parameter name</td>
<td>$F_{ba} = (m_1, m_2, r) \leftarrow \frac{g \cdot m_1 \cdot m_2}{r^2}$ \hspace{1cm} (6.47)</td>
</tr>
<tr>
<td>Using the equality operator ((=)) instead of the arrow in function definition.</td>
<td>$FF := (m_1, m_2, r) = \frac{g \cdot m_1 \cdot m_2}{r^2}$</td>
</tr>
<tr>
<td>Troubleshooting function definitions</td>
<td>( (m_1, m_2, r) = \frac{g \cdot m_1 \cdot m_2}{r^2} ) (6.48)</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>Although technically not an error, defining a function without using the arrow notation is not prescribed by these notes.</td>
<td>( F_2(m_1, m_2, r) = \frac{g \cdot m_1 \cdot m_2}{r^2} )  ( (m_1, m_2, r) \rightarrow \frac{g \cdot m_1 \cdot m_2}{r^2} ) (6.49)</td>
</tr>
</tbody>
</table>
7 Chapter 7 Programming with functions

7.1 Chapter overview

Most computer languages regard functions or procedures as the "lego blocks" for building programs. Not only is it expected that there will be a lot of different kinds of blocks that will be provided, but that you will build things by putting them several of them together. The way functions are used in this way is through *daisy chaining* -- by making the output of one function the input of another. These chains can then be defined to be functions themselves. By defining a few chains and then using them together, powerful combinations of operations can be custom-built quickly for the user's needs.

Most computer languages extend the concept of functions to go beyond the numbers or formulas that "mathematical functions" provide. In Maple, as in most of languages, a function can return other kinds of results. It is fairly common in Maple and other languages to have functions that produce as output a list, an equation, or a string as a result. As we shall see, we can even return a Maple plot as the result of a function. In a symmetric fashion, it is possible for computer functions to have lists, equations, plots, or strings as inputs.

7.2 Designing functions from context

In doing technical work, we often see functions defined as an equation relating the name of the function, its argument(s), and the function definition. Those are easy to translate into Maple's notation and use. For example if we see in a mathematics book "define f(x) = x^2 + 2x - v\theta " then we can just transcribe it into the Maple function notation:

\[ f := (x) \rightarrow x^2 + 2x - v\theta . \]

In word problems, we have to "read between the lines" and design the function. This requires answering the questions:

a) What will the inputs to the function be? Try to give symbolic names for it.

b) What will the output be? Sometimes to realize what the output is, you can create a worksheet with several steps. If there is only one final result, then that should be the output.

c) How do you calculate the output from the inputs? Hopefully, there's a simple formula that describes this.

We illustrate this process with an example:

Designing a function for pressure/temperature problems


The Ideal Gas Law, as stated in *Introduction to Engineering*) is:

\[ P \cdot V = n \cdot R \cdot T \]

\[ P V = n R T \quad (7.1) \]

where

- \( P \) is pressure in Pascals (Pa)
- \( V \) is volume in \( m^3 \).
- \( n \) is the amount of gas in moles (mol)
- \( T \) is the temperature in degrees K,

81
R is the gas constant, approximately \( 8.31 \cdot \frac{[m^2 \cdot Pa]}{[K \cdot mol]} \).

We want to solve the following problem (actually, various versions of it):

Problem

We measure the temperature and pressure of a gas. It has a pressure of 100 \( [kPa] \) and a temperature 473 degree Kelvin. Then cool it so it has a pressure of 89.8 \( [kPa] \). What is its temperature?

Finding the answer

First, we do the mathematical thinking and informal calculation that allows us to build a function that will solve all problems of this type:

For a fixed cylinder volume, according to the Ideal Gas Law:

\[
\frac{P_1}{T_1} = \frac{P_2}{T_2}, \quad \text{so} \quad T_2 = \frac{P_2}{P_1} \cdot T_1 \; \text{so} \; T_2 = \frac{89.8}{100} \cdot 473 = 424.7540000
\]

Function design

We see that the problem wants us to calculate a temperature \( T_2 \) given the atmospheric pressure \( P_1 \), the internal pressure \( P_2 \), and the first temperature, \( T_1 \).

The output is \( T_2 \), the inputs are \( P_1, P_2, \) and \( T_1 \). We are free to name the function anything we want since the problem statement doesn't name this. We decide to call it something that reminds us of the purpose.

\[
\text{secondTemp} := (P_1, P_2, T_1) \rightarrow \frac{P_2}{P_1} \cdot T_1
\]

Testing and troubleshooting the function

We see whether we get the intended result with the numbers we've already worked out. Note that if we hadn't done the analysis, we wouldn't have any way of testing what we designed.

\[
\text{secondTemp}(89.8, 100, 473) \rightarrow 526.7260579
\]
Oops, that isn't the same result. What did we do wrong? The formula 1.2.2 seems like the right thing. What else could go wrong? Close inspection indicates that the first argument to \texttt{internalTemp} is \( P_1 \), which appears in the denominator of the formula for the output. In (1.2.3), that would put the "89.8" in the denominator, but our example had 89.8 in the numerator. Oops, we gave the values in the wrong order for the function. There's nothing wrong except that we should invoke the function with the information given in the correct order:

\[
\text{secondTemp}(100, 89.8, 473)
\]

424.7540000 \hfill (7.4)

Using the function

We are given a different version of the problem:

We measure the temperature and pressure of a gas. It has a pressure of 2000 [kPa] and a temperature 473 degree Kelvin. Then cool it so it has a pressure of 53.6 [kPa]. What is its temperature?

Answer:

\[
\text{secondTemp}(2000, 56.6, 473)
\]

13.38590000 \hfill (7.5)

Since the answer is in degrees Kelvin, this is only about 14 degrees above absolute zero. That's pretty cold!

The usefulness of alternative function designs

Suppose we had this new problem:

We measure the temperature and pressure of a gas. It has a pressure of 100 [kPa] and a temperature 473 degree [K]. We then heat it to 512 degrees Kelvin. What is its pressure then?

Another function designed

A little thought produces the calculation:

\[
P_2 = \frac{P_1}{T_1} = \frac{100}{473} = 108.2452431
\]

This leads to the function definition:

\[
\text{secondPressure} := (P_1, T_1, T_2) \rightarrow \frac{P_1}{T_1} \frac{T_2}{T_2}
\]

\[
(P_1, T_1, T_2) \rightarrow \frac{P_1 T_2}{T_1}
\]

(7.6)

We test this (remembering what happened before about the order of arguments)

\[
\text{secondPressure}(100, 473, 512.0)
\]

108.2452431 \hfill (7.7)
Conclusion

To develop functions, it helps to have worked through some typical calculations interactively. Once you have realized which quantities you are starting with and named them, and have developed the formula for the calculation using those names, you can create a function definition. You can use the names given in the problem description, or you can make up names based on their purpose. Unlike mathematics, you are not limited to single letters for names of variables or names of functions. Computer programmers know that longer names are often easier to remember or understand.

7.3 Function composition: daisy-chaining functions together

In the scripts we have developed so far, we have developed a result through a sequences of actions. These sequences can often be described through functional composition -- an expression that chains together several actions. Consider the following example:

Problem

On November 1, 2007, one Euro was worth 1.002908434 US dollars. We are buying widgets that cost 30 Euros each and importing them into the US. We then put the widgets into packages that cost .075 dollars. How much should we sell the product in the US if we want a 10% markup from our costs?

Finding the solution: step 1, first do the calculation interactively.

Doing this in the style of scripts, we first assign the values to variables, and then do the calculational steps.

\[
\text{convRate} := 1.002908434
\]

\[
1.002908434
\]  \hspace{1cm} (7.8)

\[
\text{costInEuros} := 30
\]

\[
30
\]  \hspace{1cm} (7.9)

\[
\text{pkgCost} := .075
\]

\[
0.075
\]  \hspace{1cm} (7.10)

\[
\text{markupPct} := .10
\]

\[
0.10
\]  \hspace{1cm} (7.11)

\[
\text{totalCost} := \text{pkgCost} + \text{convRate} \cdot \text{costInEuros}
\]

\[
30.16225302
\]  \hspace{1cm} (7.12)

\[
\text{sellingPrice} := (1 + \text{markupPct}) \cdot \text{totalCost}
\]

\[
33.17847832
\]  \hspace{1cm} (7.13)

We foresee using this calculation several times as the conversion Rate, the manufacturing cost in Europe, and the packaging cost change. We even see that the markup might change. We can try to boil down these steps into a few functions.

Designing the solution: step 2, design functions to do the calculational steps

\[
\text{totalCostFunc} := (\text{convRate}, \text{costInEuros}, \text{pkgCost}) \rightarrow (\text{pkgCost} + \text{convRate} \cdot \text{costInEuros})
\]
(convRate, costInEuros, pkgCost) → pkgCost + convRate costInEuros

\[
\text{priceFunc} := (\text{markupPct}, \text{totalCost}) \rightarrow (1 + \text{markupPct}) \cdot \text{totalCost}
\]

\[
\text{(markupPct, totalCost)} \rightarrow (1 + \text{markupPct}) \cdot \text{totalCost}
\]

\[
\text{sellingPriceFunc} := (\text{convRate}, \text{costInEuros}, \text{pkgCost}, \text{markupPct}) \rightarrow \text{priceFunc}(\text{markupPct}, \text{totalCostFunc}(\text{convRate}, \text{costInEuros}, \text{pkgCost}))
\]

\[
\text{(convRate, costInEuros, pkgCost, markupPct)} \rightarrow \text{priceFunc}(\text{markupPct}, \text{totalCostFunc}(\text{convRate}, \text{costInEuros}, \text{pkgCost}))
\]

Note that the way that the third function \(\text{sellingPriceFunc}\) is defined, it takes the output of \(\text{totalCostFunc}\) and makes it one of the inputs to \(\text{priceFunc}\).

Testing the solution: step 3, test the building blocks in the order that they are used

We test the first two functions. After we see that they agree with our preliminary version of the calculations, we test the third function that depends on the correctness of the first two.

\[
\text{totalCostFunc}(\text{convRate, costInEuros, pkgCost})
\]

\[
30.16225302
\]

\[
\text{priceFunc}(.10,(1.3.10))
\]

\[
33.17847832
\]

\[
\text{sellingPriceFunc}(\text{convRate, costInEuros, pkgCost, markupPct})
\]

\[
33.17847832
\]

We could write a script that just used \(\text{totalCostFunc}\) and \(\text{priceFunc}\), but by designing and using a third function, we reduce the work of handling an instance of the problem to just pasting in the values for the four parameters in one line. This is probably less work than changing four lines of parameters that we have had with our previous approach to scripts.

While function composition is a succinct way of ordering many operations, its advantages are apparent only after the chain is built and tested as working correctly. It maybe easier to develop the chain as a script of assignments and then refactor -- rewrite without changing the meaning -- the script so that it uses user-defined functions to replace some of the chains of assignments.

Using the solution: step 4, present a script that defines the functions, then invoke the "answer function" repeatedly to handle various versions of the problem

<table>
<thead>
<tr>
<th>A script that uses functional composition (chaining), and its use</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Begin function definitions</strong></td>
</tr>
</tbody>
</table>
| \[
\text{totalCostFunc} := (\text{convRate, costInEuros, pkgCost}) \rightarrow (\text{pkgCost} + \text{convRate \cdot costInEuros})
\]

\[
\text{(convRate, costInEuros, pkgCost) \rightarrow pkgCost + convRate \cdot costInEuros}
\]

\[
\text{priceFunc} := (\text{markupPct, totalCost}) \rightarrow (1 + \text{markupPct}) \cdot \text{totalCost}
\]

\[
\text{(markupPct, totalCost)} \rightarrow (1 + \text{markupPct}) \cdot \text{totalCost}
\]

\[
(\text{convRate, costInEuros, pkgCost, markupPct}) \rightarrow \text{priceFunc}(\text{markupPct, totalCostFunc}(\text{convRate, costInEuros, pkgCost}))
\]

\[
\text{(convRate, costInEuros, pkgCost, markupPct)} \rightarrow \text{priceFunc}(\text{markupPct, totalCostFunc}(\text{convRate, costInEuros, pkgCost}))
\]

\[
\text{totalCostFunc}(\text{convRate, costInEuros, pkgCost})
\]

\[
30.16225302
\]

\[
\text{priceFunc}(.10,(1.3.10))
\]

\[
33.17847832
\]

\[
\text{sellingPriceFunc}(\text{convRate, costInEuros, pkgCost, markupPct})
\]

\[
33.17847832
\]
A script that uses functional composition (chaining), and its use

\[
\text{sellingPriceFunc} := (\text{convRate}, \text{costInEuros}, \text{pkgCost}, \text{markupPct}) \rightarrow \text{priceFunc}(\text{markupPct}, \text{totalCostFunc}(\text{convRate}, \text{costInEuros}, \text{pkgCost}))
\]

End function definitions

Problem solving

Version 1

On November 1, 2002, one Euro was worth \( \frac{1}{9971} = 1.002908434 \) US dollars. We are buying widgets that cost 30 Euros each and importing them into the US. We then put the widgets into packages that cost .075 dollars. How much should we sell the product in the US if we want a 10% markup from our costs?

\[
\text{sellingPriceFunc}(1.002908434, 30, 0.75, 10)
\]

\( \$33.18 \) \hspace{1cm} (7.23)

(We got the number formatted to currency by right-click->Numeric Formatting->Currency.)

Version 2

On November 1, 2007, one Euro was worth 1.4487 US dollars. We are buying widgets that cost 33 Euros each and importing them into the US. We then put the widgets into packages that cost .09 dollars. How much should we sell the product in the US if we want a 10% markup from our costs?

\[
\text{sellingPriceFunc}(1.4467, 33, 0.09, 10)
\]

\( \$52.61 \) \hspace{1cm} (7.24)

Version 3

On November 1, 2009, one Euro was worth 1.4728 US dollars. We are buying widgets that cost 35 Euros each and importing them into the US. We then put the widgets into packages that cost .10 dollars. How much should we sell the product in the US if we want a 10% markup from our costs?

\[
\text{sellingPriceFunc}(1.4728, 35, 0.10, 10)
\]

\( \$56.81 \) \hspace{1cm} (7.25)

We see that if we are interested in looking at the solution to several different versions of the problem, setting up the script as a collection of function definitions presents the problem-solving calculation only once. We can then proceed and present the several solutions through a single-line calculation. We don't have to wade through all the steps of the calculation to see the answer to the first problem, then looking through the same steps to see the answer to the second, etc.

7.4 Expressions with units of measurements: convert

Maple has facilities for converting between various English and metric units. It is useful for doing multi-step calculations because the conversions happen automatically.

In the first way of using \( \text{convert} \), one thinks of a value as implicitly expressing a number of units and wants another number expressing those number of units converted to another unit. One uses \( \text{convert}(\text{value}, \text{units}, \text{fromUnit}, \text{toUnit}) \).

Examples of unit conversion

<table>
<thead>
<tr>
<th>Examples of unit conversion</th>
<th>Note that the answer to this was expressed as a floating point number rather than a fraction because the input was floating point (1.0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>How many inches in a meter? How many feet in a kilometer? How many millimeters in a mile?</td>
<td></td>
</tr>
</tbody>
</table>


Examples of unit conversion

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>convert(1, units, inch, meter)</code></td>
<td>$\frac{127}{5000}$</td>
</tr>
<tr>
<td><code>convert(1, units, ft, km)</code></td>
<td>$\frac{381}{1250000}$</td>
</tr>
<tr>
<td><code>convert(1.0, units, mile, mm)</code></td>
<td>$1.6093440 \times 10^9$</td>
</tr>
<tr>
<td><code>convert(5.4, units, kilowatt, horsepower)</code></td>
<td>7.241519284</td>
</tr>
<tr>
<td><code>convert(2.0, units, angstroms, micrometers)</code></td>
<td>0.000200000000</td>
</tr>
<tr>
<td><code>convert(15.0, units, miles, meters / hour / second)</code></td>
<td>6.705600000</td>
</tr>
<tr>
<td><code>convert(13.3, units, gallons / yards^3 * meter^3)</code></td>
<td>65.85005144</td>
</tr>
</tbody>
</table>

Maple can convert between most compatible units.

Sometimes units are expressed as ratios of other units. Maple can handle such conversions as well.

In many examples in the Maple documentation, some of the arguments to `convert` are quoted -- surrounded by apostrophes -- to prevent evaluation from using the value of the names of the units. For example, if you have assigned a value to the variable `s`, then you cannot convert to seconds with this name without quotation.

**Troubleshooting unit conversion**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>seconds := convert(3, units, days, seconds)</code></td>
<td>As long as the various names used as arguments to the <code>convert</code> function don't have values, things work fine.</td>
</tr>
<tr>
<td><code>seconds2 := convert(4, units, minutes, seconds)</code></td>
<td>Maple performs evaluation of names as it figures out what the inputs to <code>convert</code> is. Since <code>seconds</code> has a value, Maple tries to compute <code>convert(4, units, minutes, 259200)</code>. Since the 4th argument to <code>convert</code> has to be a name, an error results.</td>
</tr>
<tr>
<td><code>seconds2 := convert(4, units, minutes, 'seconds')</code></td>
<td>Quoting the 4th argument causes the name <code>seconds</code> to be given as the 4th input to <code>convert</code>. This works.</td>
</tr>
<tr>
<td><code>seconds3 := convert(5, 'units', 'hours', 'seconds')</code></td>
<td>Quoting all the names as a prophylactic measure is acceptable. You see this in a lot of the Maple on-line documentation.</td>
</tr>
<tr>
<td><code>Error, (in convert/units) unable to convert 'min' to '259200'</code></td>
<td>This is the error message you see when you are trying to convert between incompatible units, e.g. trying to convert a gallon into a meter. <code>pc</code> seems to be Maple's internal name for parsec, <code>d</code> the name for days.</td>
</tr>
</tbody>
</table>

In *Star Wars Episode IV: A New Hope*, Han Solo says that the Millenium Falcon made the Kessel Run in "less than twelve parsecs". We want to know how many days a parsec is.
### Troubleshooting unit conversion

<table>
<thead>
<tr>
<th>Equation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>convert(12.0, units, parsecs, days)</td>
<td>Error, (in convert/units) unable to convert ‘pc’ to ‘d’. (7.37)</td>
</tr>
<tr>
<td>convert(12.0, units, parsecs, miles)</td>
<td>2.300821388 \times 10^{14} (7.38) A parsec is a non-fictional unit of distance, not time, so we can convert 12 parsecs to miles, kilometers, inches... But we can't convert it to days any more than we can convert inches to volts.</td>
</tr>
</tbody>
</table>

### A problem solved, a script built using function definitions

<table>
<thead>
<tr>
<th>Equation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A car travels a 45 miles per hour. How many minutes does it take to travel 900 kilometers?</td>
<td>We build a sequence of calculations to understand how to solve this problem. This is the informal phase of development, while we are trying to understand what to do. Once we have an idea, we start designing functions and testing them.</td>
</tr>
</tbody>
</table>

\[
distance := 900.0
\]

\[
speed := 45
\]

\[
d := convert(distance, units, kilometers, miles)
\]

\[
t := \frac{d}{speed}
\]

\[
convert(t, units, hours, minutes')
\]

<table>
<thead>
<tr>
<th>Calculation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>distance := 900.0</td>
<td>900.0 (7.39)</td>
</tr>
<tr>
<td>speed := 45</td>
<td>45 (7.40)</td>
</tr>
<tr>
<td>d := convert(distance, units, kilometers, miles)</td>
<td>559.2340730 (7.41)</td>
</tr>
<tr>
<td>t := \frac{d}{speed}</td>
<td>12.42742384 (7.42)</td>
</tr>
<tr>
<td>convert(t, units, hours, minutes')</td>
<td>745.6454304 (7.43)</td>
</tr>
</tbody>
</table>

It would pretty obvious how to make a script out of this to handle any problem of the form: A car travels \( speed \) miles per hour. How many minutes does it take to travel \( distance \) kilometers... With a few user defined functions, we can get the answer with less typing/cutting/pasting

\[
dConvert := (distance) \rightarrow convert(distance, units, kilometers, miles)
\]

\[
dConvert(900)
\]

<table>
<thead>
<tr>
<th>Value</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>781250</td>
<td>1397 (7.45)</td>
</tr>
</tbody>
</table>

\[
tCalc := (d, speed) \rightarrow \frac{d}{speed}
\]

\[
tCalc((1.4.18), 45)
\]

<table>
<thead>
<tr>
<th>Value</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>156250</td>
<td>12573 (7.47)</td>
</tr>
</tbody>
</table>

The first step was to convert the distance from kilometers to miles. We create a function that does it. We test it to the result that we got in the script above and see that it agrees.

The next step was to calculate the time (in hours) from the distance in miles and the speed in mph. The test shows that tCalc seems to be built correctly.
A problem solved, a script built using function definitions

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>( tConv := ( t ) \rightarrow convert(t, \text{units}, \text{hours}', \text{minutes}' ) )</td>
<td>( t \rightarrow convert(t, \text{units}, \text{hours}', \text{minutes}' ) )</td>
<td>The third step was to convert the time from hours to minutes. The test of this step agrees with what the script does, too.</td>
</tr>
<tr>
<td>( tConv(1.415) )</td>
<td>( \frac{3125000}{4191} )</td>
<td>(7.48)</td>
</tr>
<tr>
<td>( solveIt := ( \text{speed}, \text{distance} ) \rightarrow tConv( \text{tCalc}(dConvert(\text{distance}), \text{speed}) ) )</td>
<td></td>
<td>(7.49)</td>
</tr>
<tr>
<td>( \text{solveIt}(45, 900.0) )</td>
<td>( 745.6454304 )</td>
<td>(7.50)</td>
</tr>
</tbody>
</table>

The above table showed the thinking behind the design and testing of the multi-step calculation. However, in "what we would hand in", we don't show the testing or the initial script, just the definition of the functions, and then the repeated invocation of the "solution function". Define the functions once, then invoke it repeatedly. This eliminates the need for repeated cutting/pasting/selection for execution.

Solving multiple versions of a problem through functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>( dConvert := ( \text{distance} ) \rightarrow convert(\text{distance}, \text{units}, \text{kilometers}, \text{miles} ) )</td>
<td>( \text{distance} \rightarrow convert(\text{distance}, \text{units}, \text{kilometers}, \text{miles} ) )</td>
<td>The travelSoln function indicates the parameters of the script, speed and distance. In setting up the solution method as a function, we lose some intelligibility because of the &quot;inside out&quot; style of following the operation of daisy-chained function composition. However, we gain convenience using the script. Using this form makes it easier to see several solutions, because you don't have to wade through all the lines of script that work out the answer, just a single line setting up the values of the parameters and printing out the answer.</td>
</tr>
<tr>
<td>( tCalc := ( d, \text{speed} ) \rightarrow \frac{d}{\text{speed}} )</td>
<td>((d, \text{speed}) \rightarrow \frac{d}{\text{speed}} )</td>
<td>(7.52)</td>
</tr>
<tr>
<td>( tConv := ( t ) \rightarrow convert(t, \text{units}', \text{hours}', \text{minutes}' ) )</td>
<td>( t \rightarrow convert(t, \text{units}', \text{hours}', \text{minutes}' ) )</td>
<td>(7.53)</td>
</tr>
<tr>
<td>( travelSoln := ( \text{speed}, \text{distance} ) \rightarrow tConv( \text{tCalc}(dConvert(\text{distance}), \text{speed}) ) )</td>
<td></td>
<td>(7.54)</td>
</tr>
<tr>
<td>( \text{travelSoln}(45, 900.0) )</td>
<td>( 745.6454304 )</td>
<td>(7.55)</td>
</tr>
</tbody>
</table>

Problem version A

A car travels at 45 miles per hour. How many minutes does it take to travel 900 kilometers?

\( \text{travelSoln}(45, 900.0) \)

Problem version B
Solving multiple versions of a problem through functions

A car travels at 45 miles per hour. How many minutes does it take to travel 452 kilometers?

\[
\text{travelSoln}(45, 452.0) = 374.4797052
\]

(7.57)

Problem version C

A car travels at 65 miles per hour. How many minutes does it take to travel 1500 kilometers?

\[
\text{travelSoln}(65, 1500.0) = 860.3601126
\]

(7.58)

For casual unit conversion, it can still be useful to rely upon Maple's encyclopaedic knowledge of how to convert units. You can access this through Tools->Assistants->Unit Calculator

### 7.5 Inputs and outputs to user-defined functions don't have to be numbers

Although you don't see much mention of this in mathematics texts, it is fairly common while programming to define and use functions that take inputs and produce outputs that are not numbers. For example, if we have a list \( L \) of numbers, we can create a function that takes a list as input and produces the average of all the numbers as its output.

#### A function that takes a list as its input

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
</table>
| \[
\text{average} := L \rightarrow \frac{\sum_{i=1}^{\text{nops}(L)} L[i]}{\text{nops}(L)}
\]   | We are expecting the input to be a list of numbers. As explained in chapter 4, \( L[i] \) uses indexing to get the \( i \)-th value of the list. (See section 4.2.) \( \text{nops}(L) \) is the number of elements in the list. |
| \[
\text{average}([5, 7, -3, 2, 6])
\]   | When we invoke the function, \( L \) is \([5,7,-3,2,6]\), so \( \text{nops}(L) \) is 5. Since at least one of the elements of the list was a limited precision number (5. has a decimal point), the limited precision arithmetic is performed with it and subsequent steps of the sum. |
A function that takes a list as its input

| 3.375000000 |

(7.61)

In analyzing mathematical models as we have been doing, it is also useful to produce abbreviations for common combinations of plot options by creating a function that produces a plot as its result.

A function that returns a plot as its output, rather than a number

Problem

We are given two lists of data, \( pData \), and \( tData \). Plot \( pData \) as a function of \( tData \), and vice versa.

Solution

Build a function that becomes an abbreviation for the operations in the plot. Provide a third argument that is the string for the color.

\[
PlotIt := (xData, yData, c, L) \rightarrow \text{plot}(xData, yData, style = point, color = c, labels = L)
\]

\[
(\text{xData}, \text{yData}, \text{c, L}) \rightarrow \text{plot}(\text{xData}, \text{yData}, \text{style = point, color = c, labels = L})
\]  

(7.62)

\[
pData := [134.2, 142.5, 155.0, 159.8, 171.1, 184.2];
\]

[134.2, 142.5, 155.0, 159.8, 171.1, 184.2]  

(7.63)

\[
tData := [0, 20.1, 39.8, 60.0, 79.9, 100.3];
\]

[0, 20.1, 39.8, 60.0, 79.9, 100.3]  

(7.64)

Plot pressure versus temperature, in red. Note: there seems to be a bug in Maple that suppresses the printing of the horizontal axis label.

\[
PlotIt(pData, tData, "red", ["pressure", "temperature"])
\]

Plot temperature versus pressure, in red. Copying and pasting the invocation of the PlotIt function we defined is easier than changing the insides of the original plot operation.

\[
PlotIt(tData, pData, "blue", ["temperature", "pressure"])
\]
It is possible to return a list or sequence as a result of a function. Such a function can be put in a chain.

A problem solved with a function that outputs a sequence of two numbers

Problem A

A right triangle has a hypotenuse of length 5 feet. The angle between it and one of its sides is 10 degrees. What is the length of the perimeter?

We first build a function that computes the two sides of the right triangle and returns the two values as a sequence. We have to convert degrees into radians in order to do this because the Maple trig functions all use radians.

\[
\text{sideSide} := (\text{hypo}, \text{angle}) \rightarrow (\text{hypo} \cdot \sin(\text{convert(\text{angle} \cdot \text{degrees}, \text{radians}))}, \text{hypo} \cdot \cos(\text{convert(\text{angle} \cdot \text{degrees}, \text{radians}))})
\]

(7.65)

Let's test the sideSide function.

\[
\text{sideSide}(5, 10.0)
\]

\[
5 \sin(0.055555555556 \pi), 5 \cos(0.055555555556 \pi)
\]

(7.66)

Now, develop a function that takes a sequence of three numbers and adds them together.

\[
\text{sumSides} := (a, b, c) \rightarrow a + b + c
\]

(7.67)

By chaining together the output of sideSide and making it part of the input of sumSides, we can get the whole computation done in one function.

\[
\text{perimeter} := (\text{hypo}, \text{angle}) \rightarrow \text{sumSides}(\text{sideSide}(\text{hypo}, \text{angle}), \text{hypo})
\]

(7.68)

\[
\text{perimeter}(5, 10)
\]

\[
5 \sin\left(\frac{1}{18} \pi\right) + 5 \cos\left(\frac{1}{18} \pi\right) + 5
\]

(7.69)

\[
10.792
\]

(7.70)
A problem solved with a function that outputs a sequence of two numbers

**Problem B**

A right triangle has a hypotenuse of length 10 feet. The angle between it and one of its sides is 42 degrees. What is the length of the perimeter?

\[
\text{evalf(perimeter}(10, 42))
\]

\[24.12275432\] (7.71)

Once we have done the work to design and test the functions out on a problem, we can present a script that can solve several different versions of the problem:

<table>
<thead>
<tr>
<th>Solving several versions of a function with function definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin function definitions</td>
</tr>
<tr>
<td>A function that computes the two sides of a right triangle given the angle and the length of the hypotenuse</td>
</tr>
</tbody>
</table>

\[
sideSide \equiv (\text{hypo}, \text{angle}) \rightarrow (\text{hypo} \cdot \sin(\text{convert(\text{angle} \cdot \text{degrees}, \text{radians})})), \text{hypo} \cdot \cos(\text{convert(\text{angle} \cdot \text{degrees}, \text{radians})}))
\]

\[
(hypo, \text{angle}) \rightarrow (\text{hypo} \cdot \sin(\text{convert(\text{angle} \cdot \text{degrees}, \text{radians})})), \text{hypo} \cdot \cos(\text{convert(\text{angle} \cdot \text{degrees}, \text{radians})}))
\] (7.72)

A function that takes a sequence of three numbers and adds them together.

\[
\text{sumSides} \equiv (a, b, c) \rightarrow a + b + c
\]

\[
(a, b, c) \rightarrow a + b + c
\] (7.73)

Compute the perimeter by summing the three sides.

\[
\text{perimeter} \equiv (\text{hypo}, \text{angle}) \rightarrow \text{sumSides(sideSide(hypo, angle), hypo)}
\]

\[
(hypo, \text{angle}) \rightarrow \text{sumSides(sideSide(hypo, angle), hypo)}
\] (7.74)

End function definitions

**Problem A Solution**

A right triangle has a hypotenuse of length 5 feet. The angle between it and one of its sides is 10 degrees. Approximately, what is the length of the perimeter in feet?

\[
\text{evalf(perimeter}(5, 10))
\]

\[10.79227965\] (7.75)

**Problem B Solution**

A right triangle has a hypotenuse of length 10 feet. The angle between it and one of its sides is 42 degrees. Approximately, what is the length of the perimeter in feet?

\[
\text{evalf(perimeter}(10, 42))
\]

\[24.12275432\] (7.76)
## 7.6 Chapter Summary

<table>
<thead>
<tr>
<th>Function design</th>
<th>a) What will the inputs be?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Designing functions from context</td>
<td>b) What will the output be?</td>
</tr>
<tr>
<td></td>
<td>c) How do we calculate the output from the inputs?</td>
</tr>
</tbody>
</table>

| Function composition                  | $A := (x, y) \rightarrow \frac{1}{x} + 3 \cdot x^3 + 3 \cdot y$ |
|                                      | $(x, y) \rightarrow \frac{1}{x} + 3 \cdot x^3 + 3 \cdot y$  (7.77) |
|                                      | $B := (x, y) \rightarrow \frac{3}{A(x,y)}$ |
|                                      | $(x, y) \rightarrow \frac{3}{A(x,y)}$  (7.78) |
| $B(3, 1)$                             | $\frac{9}{253}$  (7.79) |

<table>
<thead>
<tr>
<th>Unit conversion</th>
<th>convert(1, units, inch, meter)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\frac{127}{5000}$  (7.80)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compound units expressed in ratio form can also be converted into compatible compound units.</th>
<th>convert(15.0, units, miles, hour, second)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$6.705600000$  (7.81)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Converting between incompatible units will throw an error.</th>
<th>convert(3, units, days, miles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Error, (in convert/units) unable to convert 'd' to 'mi'  (7.82)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>If we create a variable with the same name as a unit, trying to convert using the variable name will throw an error.</th>
<th>seconds := 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$5$  (7.83)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>To avoid this, we can use single quotes around the unit names to specify using the unit string, as opposed to the variable value.</th>
<th>convert(3, units, 'days','seconds')</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Error, (in convert/units) unable to convert 'd' to '5'  (7.84)</td>
</tr>
<tr>
<td></td>
<td>convert(3, units,'days','seconds')</td>
</tr>
<tr>
<td></td>
<td>$259200$  (7.85)</td>
</tr>
</tbody>
</table>

| We can use Maple's built-in unit converter to convert units using drop-down menus. | This tool is located in Tools>Assistants>Unit Calculator |

<table>
<thead>
<tr>
<th>Non-number inputs and outputs of a function</th>
<th>Using a list of numbers as an input</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\text{average} := L \rightarrow \frac{\sum_{i=1}^{\text{nops}(L)} L[i]}{\text{nops}(L)}$</td>
</tr>
</tbody>
</table>

### Function composition

- **Function**: $A := (x, y) \rightarrow \frac{1}{x} + 3 \cdot x^3 + 3 \cdot y$
  
  - **Input**: $(x, y)$
  - **Output**: $\frac{1}{x} + 3 \cdot x^3 + 3 \cdot y$
  
- **Function**: $B := (x, y) \rightarrow \frac{3}{A(x,y)}$
  
  - **Input**: $(x, y)$
  - **Output**: $\frac{3}{A(x,y)}$
  
- **Evaluation**: $B(3, 1) = \frac{9}{253}$
### Non-number inputs and outputs of a function

\[ \sum_{i=1}^{\text{nops}(L)} L_i \]

\[ L \rightarrow \frac{\text{nops}(L)}{\text{nops}(L)} \]

\[ \text{average}([5, 7, -3, 2, 6]) \]

\[ \frac{17}{5} \]

### Returning a plot instead of a number

\[ \text{PlotIt} := (x \text{Data}, y \text{Data}, c, L) \rightarrow \text{plot}(x \text{Data}, y \text{Data}, \text{style} = \text{point}, \text{color} = c, \text{labels} = L) \]

\[ (x \text{Data}, y \text{Data}, c, L) \rightarrow \text{plot}(x \text{Data}, y \text{Data}, \text{style} = \text{point}, \text{color} = c, \text{labels} = L) \]

\[ \text{PlotIt}([1, 2, 3], [2, 1, 2, 1], \text{\textcolor{blue}{blue}}, \text{\textcolor{black}{[\text{x}, \text{y}]}]) \]

![Plot](image)

### Returning a sequence of numbers instead of a single number

\[ \text{sideSide} := (\text{hypo}, \text{angle}) \rightarrow (\text{hypo sin} (\text{convert(angle degrees, radians)}), \text{hypo cos} (\text{convert(angle degrees, radians)})) \]

\[ (\text{hypo}, \text{angle}) \rightarrow (\text{hypo sin} (\text{convert(angle degrees, radians)}), \text{hypo cos} (\text{convert(angle degrees, radians)})) \]

\[ \text{sideSide}(1, 45) \]

\[ \frac{1}{2} \sqrt{2}, \frac{1}{2} \sqrt{2} \]
8 Chapter 8 Visualization, modeling, and simulation

8.1 Chapter Overview

1. The purpose of with -- loading packages so that you can access additional operations in Maple -- is illustrated with curve fitting and plotting.

2. Plot structures are the result of the plot operation. They can be assigned to variables through \( := \) just as numbers, formulas, lists, function definitions, etc. can be.

3. The display operation of the plots package is explained. Typically, before display is used, the user issues a with command, and assigns some variables plot structures that are mentioned in the display operation. The display operation can be used to generate more sophisticated and enlightening pictures of phenomena.

4. Much of our computational work to date has been for the purpose of answering questions that can be explored with the help of mathematical models. With the appropriate mathematics, personal computer or supercomputer-class calculations can be used to come up with reasonably accurate descriptions of phenomena. Mathematical models are used in modern engineering because the "build it and see" methodology often seen in elementary student work is not cost-effective once one moves about beyond simple, low-cost situations.

5. Simulation is the art of computationally predicting the behavior of system entities as they change over time, through the use of mathematical models. It can be as simple as using functions that, given the time \( t \) as input, calculate the position, size, weight, or other state of a system entity.

5. The animate operation of the plots package is explained. Its use is illustrated with a session of question-answering using a mathematical models of moving bodies. computer-generated animations are another useful tool besides solve, and plot.

8.2 Using functions from packages, with

Although we have seen dozens of built-in functions so far in Maple, there are several thousand more. Some of them are defined in your Maple program when it starts up. However, it is not done for most of the built-in functions. There are so many that if that were done for all of them, Maple would take a long time to start up and would require large amounts of memory even before you had done any work in it.

Most built-in functions are organized into collections called packages. The general way to access a function belonging to a package is through \( \text{package}[\text{function}] \). The least squares function belongs to a package named CurveFitting, hence its name is CurveFitting[LeastSquares]. The with(package name) operation in Maple will load all the functions in the specified package into Maple. After this operation, functions can be referred to with just their "short name", e.g. LeastSquares rather than CurveFitting[LeastSquares]. Doing a with(package) can save you typing if you expect to use a function from a package several times during a Maple session. Ending the line with a colon (:) will suppress printing of all the functions in the package that usually occurs.

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>with restart</td>
<td>The CurveFitting package has a number of functions for data fitting. One is called LeastSquares. However, LeastSquares(...) does nothing. Although there is no error message, this is a mistake -- we didn't load the package in with a &quot;with&quot; so the least squares data fitting function isn't known. We get the same kind of behavior as if we had typed in ( f(1,2,3) ) with ( f ) undefined -- Maple just spits back what we typed in.</td>
</tr>
</tbody>
</table>

\[
\text{pData} := [134.2, 142.5, 155.0, 159.8, 171.1, 184.2];
\]
\[
[134.2, 142.5, 155.0, 159.8, 171.1, 184.2]
\]

\[
\text{tData} := [0, 20.1, 39.8, 60.0, 79.9, 100.3];
\]
\[
[0, 20.1, 39.8, 60.0, 79.9, 100.3]
\]
Example | Commentary
--- | ---
\[
\text{pressureFormula := } \text{LeastSquares}(tData, pData, t) \\
\quad \text{LeastSquares}\{[10, 20.1, 39.8, 60.0, 79.9, 100.3], [134.2, 142.5, 155.0, 159.8, 171.1, 184.2], t) \\
\]
(8.3) | Doing a "with" gives the names of all the functions in the package.

\[
\text{with}(	ext{CurveFitting}), \\
\quad [\text{ArrayInterpolation, BSpline, BSplineCurve, Interactive, LeastSquares, PolynomialInterpolation, RationalInterpolation, Spline, ThieleInterpolation}] \\
\]
(8.4) | Once we do the \text{with}, \text{LeastSquares} works.

\[
\text{pressureFormula := } \text{LeastSquares}(tData, pData, t) \\
\quad 133.5000490 + 0.4858370741 \, t \\
\]
(8.5) | We can fit a line to temperature as a function of pressure.

\[
\text{tempFormula := } \text{LeastSquares}(pData, tData, p) \\
\quad -271.0549580 + 2.034674428 \, p \\
\]
(8.6) |

### 8.3 plot structures, display and combining plots

Like \text{solve}, Maple plot is a function: it has inputs and produces outputs. What kind of output does the plot function produce? In Maple, the result of \text{plot} is a special type of result called a plot structure. What happens when you evaluate an expression in Maple that invokes the plot function is that a plot structure is created. If the plot structure is then assigned to a variable (through :=, for example), then an ellipsis of the plot structure is displayed. If the plot structure is the entire result and there is no assignment, then the plot is displayed.

#### plot results, displayed and not displayed

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
</table>
| \[
\text{plot}(\sin(x), x = 0 \ldots 10) \\
\]
| Evaluating an expression and then displaying the result. |

| \[
\quad p := \text{plot}(\cos(x), x = 0 \ldots 10) \\
\quad \text{plot}(...) \\
\]
| Evaluating an expression and then assigning it to the name \text{p}. This does not display the plot. We just see \text{PLOT(…)} which is a sign that the value of \text{p} is a plot structure. |

| \[
\quad p \\
\]
| Evaluating an expression -- just \text{p} -- causes the result (the plot) to be displayed. |
The display function from the plots package takes as its first argument a list of plot structures. It will produce a plot structure that combines all the plots together. \textit{display} is the way to get a multi-plot in a script without doing cutting and pasting of plots.

\textbf{displayplots}

<table>
<thead>
<tr>
<th>display combines plots</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{timeData := [4.5, 5.5, 6, 6.5, 7, 7.5, 8, 8.5, 9, 9.5, 10, 10.5, 11, 11.5, 12, 12.5, 13]}</td>
</tr>
<tr>
<td>\texttt{tempData := [58, 57.5, 57, 56, 55.5, 54, 53, 52, 51, 50, 49, 48.5, 48.5, 47.5, 47.5, 46, 45.5, 45]}</td>
</tr>
<tr>
<td>\texttt{plot(timeData, tempData, style = point, color = &quot;red&quot;)}</td>
</tr>
<tr>
<td>\texttt{PLOT(\ldots)}</td>
</tr>
<tr>
<td>(8.10)</td>
</tr>
<tr>
<td>\texttt{formula := 25.0 + 33.61425396e^{-0.0617239065 t + 0.2777548079}}</td>
</tr>
<tr>
<td>\texttt{25.0 + 33.61425396e^{-0.0617239065 t + 0.2777548079}}</td>
</tr>
<tr>
<td>(8.11)</td>
</tr>
<tr>
<td>\texttt{plot(formula, t = \text{min}(timeData), \text{max}(timeData), color = &quot;blue&quot;)}</td>
</tr>
<tr>
<td>\texttt{PLOT(\ldots)}</td>
</tr>
<tr>
<td>(8.12)</td>
</tr>
<tr>
<td>\texttt{with(plots)} :</td>
</tr>
<tr>
<td>\texttt{display([p1, p2])}</td>
</tr>
</tbody>
</table>

\texttt{p1} is assigned a plot structure, the point plot of the data points. \texttt{p2} is assigned another plot structure, the smooth plot of the formula.

\texttt{with(plots)}:

\texttt{display([p1, p2])}
display combines plots

pose them together, as if they had been copied and pasted. This is the way to have a multi-plot in a script.

```
vertLine := line([[12, 25], [12, 58]], color = "green")
CURVES([[12, 25], [12, 58]], COLOUR(RGB, 0., 1.0000000, 0.))
```

(8.13)

```
display([p1,p2,vertLine])
```

The plottool package has a number of functions that create plot structures. One of them, line, will draw a line segment with a specified color. You can look up the details by reading the on-line documentation of the plottools and clicking on the link for line.

We create a plot structure that is a green line segment running between the points (12,25), and (12, 58).

We can create a picture with the two plots and the vertical green line. This highlights the value of the curve at t=12 minutes.

8.4 plottools, lines, and other shapes

The plottools package has a number of functions that are useful for inclusion in visualizations (plots). For example, you can create a line segment of any desired color and line thickness with the line function.

One can then use the display function to merge together lines, plots, and other shapes.

The on-line documentation on plottools contains links to further description.

: lines and other shapes: a frivolous drawing

We create a bit of "modern art" by drawing a circle and a point plot.

```
with(plottools);
```
As the on-line documentation indicates, the first two arguments to `line` are lists indicating the coordinates of the starting point and ending point of the line segment. There are optional arguments that indicate color and line thickness, etc.

\[
\text{\texttt{l1 := line([[0, 1], [1, 1]], \textcolor{orange}{\texttt{color = "orange"}});}}
\]

\[
\text{\texttt{CURVES([[[0, 1], [1, 1]], \texttt{COLOUR(RGB, 0.80000000, 0.19607843, 0.19607843)]]}}
\]

\[
\text{\texttt{l2 := line([[1, 1], [1, 2]], \textcolor{blue}{\texttt{color = "blue", thickness = 20}});}}
\]

\[
\text{\texttt{CURVES([[[1, 1], [1, 2]], \texttt{COLOUR(RGB, 0., 0., 1.00000000), THICKNESS(20)]])}}
\]

We suppress printing of the plot structure CURVES(...) for c3 with a colon because it's long and we don't want to see it, we want to see the picture it describes. Note that Maple exposes its Canadian roots by using "colour".

\[
\text{\texttt{c3 := circle([2, 2.5], 3, \textcolor{Purple}{\texttt{color = "Purple", thickness = 5}) :}}}
\]

\[
\text{\texttt{with(plots):}}
\]

\[
\text{\texttt{display([\texttt{l1, l2, c3}, \texttt{plot(2-r^2, t = 0..2, style = point, symbol = circle, symbolsize = 30)})]}}
\]

One of the options to plot (and display) is to not show the axes. Another option is to indicate that the scaling should be constrained to be equivalent in both horizontal and vertical directions (to make the circle look like a circle). That gives us an unframed work of art!

\[
\text{\texttt{display([\texttt{l1, l2, c3}, \texttt{plot(2-r^2, t = 0..2, style = point, symbol = circle, symbolsize = 30)}, axes = none, scaling = constrained])}}
\]
8.5 Mathematical models

Mathematical models try to describe a "real" situation in terms of equations and formulae. The point of modeling is to try, through mathematical or computational means, to determine what will happen without having to run experiments in the "real" situation. As Dr. Jay Brockman of the University of Notre Dame says in his book *Introduction to Engineering*:

Some engineering students have been fortunate enough to participate in pre-engineering programs such as the first LEGO(TM) League robotics design competition or American Society of Civil Engineering bridge-building contests. In addition to fostering creative problem-solving skills, such projects also introduce students to the important notion that seemingly good ideas don't always work out in practice. Often in such programs, students have ample opportunity to test and modify their designs before they formally evaluate them. If the design doesn't work, then like a sculptor working with clay, the designer adds something here or removes something there until the design is acceptable.

This cut-and-try methodology is also sometimes used in industry, particularly in circumstances where the design is simple, or where, the risk or cost of failure is low. In many situations, however, there is no second chance in the event of failure. For engineering systems such as buildings, bridges, or airplanes -- top name just a few -- failure to meet specifications could mean a loss of life. For others -- such as the integrated circuit chip -- the cost of fabrication is so high that a company may not be able to afford a second chance. In these situations, it's critical for the engineering team to be highly confident that a design will be acceptable before it's built. To do this, engineers use models to predict the behavior of their designs. A model is an approximation to a real system, such that when actions are performed on the model, it will respond in a manner similar to the real system. Models can have many different forms, ranging from physical prototypes such as a crash-test dummy to complex computer simulations.

We can think of a mathematical model as a kind of virtual system... whose input is a set of variables that represent either aspects of the design or aspects of the environment, and whose output is a set of variables that represent the behavior of the system. In side is a set of mathematical relationships that describe the operation of the system.


The point of expressing a situation mathematically is to use mathematics and computation to better understand the situation. Usually we are given or derive a formulas that allow us to calculate key properties of entities in the system. For models involving only a few variables, this can involve the following kinds of actions:

1. Get a single number, by evaluating a formula or function.
2. Get a single number, by solving an equation.
3. Gain an understanding of a relationship between one or more entities of interest and the "input variables" by producing a formula.
4. Gain a visual understanding of the relationship by plotting a function, or possibly several plots merged together.
For example, in the *(page 81)* the mathematical model is the formula that expresses the relationship between temperature and pressure. The formula was produced through a prior data-fitting computation (not shown in the example, but it took several steps and used CurveFitting[LeastSquares] eventually) -- computation type (3). We can use the relationship to calculate pressure given temperature through evaluating the formula -- computation type (1). We can use `solve` to find the temperature corresponding to a specified pressure -- computation type (2). We can see the exponential nature of the relationship, and how well the formula fits the data by plotting the point plot of the data and the plot of the formula together -- computation type (4).

### 8.6 Drawing x-y position as a function of time through parameterized plots

In simulation, the mathematical model can sometimes describe the position of a system entity as a function of time. If the entity's position is two dimensional, then we have two functions, often called \( x(t) \) and \( y(t) \). We can generate a plot of position for various values of \( t \) with a special form of `plot`.

\[
\text{plot( [ x-position expression, y-position expression, var = low..high], plot options) }
\]

will draw a two dimensional graph connecting the (x,y) points traced out for the values of the expression as the variable `var` takes on values between `low` and `high`.

#### Examples of plots where the x and y positions are expressions parameterized by t.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_{pos} := (t) \rightarrow 3 \cdot \cos(t) )</td>
<td>( t \rightarrow 3 \cos(t) )</td>
</tr>
<tr>
<td>( y_{pos} := (t) \rightarrow 2 \cdot \sin(t) )</td>
<td>( t \rightarrow 2 \sin(t) )</td>
</tr>
</tbody>
</table>

We do this plot with the horizontal and vertical axes using equivalent scaling. If we don't do this, the ellipse will look more like a circle.

\[
\text{plot([xpos(t), ypos(t), t = 0..2\cdot\pi], scaling = constrained)}
\]

![Plot of parameterized expressions](image)

**Example 2**

In this example, we have parameterized expressions for an object shot out of a cannon with horizontal velocity 10 feet/second and vertical velocity 10 feet/second minus the acceleration due to gravity.

\[
x_{pos^2} := (t) \rightarrow 10 \cdot t
\]

\( t \rightarrow 10 \cdot t \)
Examples of plots where the $x$ and $y$ positions are expressions parameterized by $t$.

\[ y_{pos2} := (t) \rightarrow 10 \cdot t - \frac{32 \cdot t^2}{2} \quad t = 10 \cdot t - 16 \cdot t^2 \tag{8.20} \]

plot([xpos2(t), ypos2(t), t = 0 .. .5], scaling = constrained, color = "blue", labels = ["x", "y"])  

\[ \text{Examples of plots where the } x \text{ and } y \text{ positions are expressions parameterized by } t. \]

\[ x(t) = 50 \cdot \sin\left( \frac{\pi}{4} \right) \cdot t \]
\[ y(t) = 50 \cdot \cos\left( \frac{\pi}{4} \right) \cdot t - 16 \cdot t^2 \]

---

8.7 Animations (movies) using animate

To the four computational modeling operations of the previous section, we add a new one:

5. Gain an understanding of a relationship of how a system changes over time. If, given a value for time $t$, we can calculate the interesting properties of the entity (e.g. its position, velocity, weight, size, color, etc.), then we can simulate that entity as it changes during a period of time. In some cases we can better understand those changes by producing a computer-generation animation (movie) of it.

The \textit{animate} operation of the \textit{plots} package has this general form:

\begin{verbatim}
How to invoke the \textit{animate} operation

\texttt{with(plots):}
\texttt{animate( function that produces a plot, [ sequence of parameters to function], time variable = low .. high)}
\end{verbatim}

\texttt{animate} has many options. See the on-line documentation and the examples in this section for further details.

\begin{verbatim}
Example

The \textit{function that produces a plot}, can be \texttt{plot} itself:

\texttt{animate(plot([t],[t^2], style=point,color="Purple",t=0..10)}

This produces a movie of a point moving through the points (0,0), (.1, .01), (.2, .04), etc. up to (10, 100).

Creating an animation of a moving object

An object shot in the air at a 45 degree angle at a speed of 50 feet per second has its (x,y) position described by the following formulae:

\[ x(t) = 50 \cdot \sin\left( \frac{\pi}{4} \right) \cdot t \]
\[ y(t) = 50 \cdot \cos\left( \frac{\pi}{4} \right) \cdot t - 16 \cdot t^2 \]
Draw an animation that shows a point moving according to these rules, for \( t=0 \) to \( t=2 \).

Solution

We define some functions for position as given by the problem.

\[
x := (t) \rightarrow 50 \cdot \sin\left(\frac{\pi}{4}\right) t
\]

\[
y := t \rightarrow 50 \cos\left(\frac{1}{4} \pi \right) t - 16 t^2
\]

We create the function that animate wants. Given \( t \) as input, it draws a point plot of a circle at the desired position.

\[
\text{posPlot} := (t) \rightarrow \text{plot}([x(t)], [y(t)], \text{style} = \text{point}, \text{symbol} = \text{circle})
\]

Now we invoke animate with this function.

\[
\text{animate}(\text{posPlot}, [t], t = 0..2)
\]

Clicking on this plot will bring up the animation tool bar, described in the next section. The tool bar can be used to play the animation.

### 8.8 Controlling animations with the animation tool bar

Clicking on the animation will display animation controls on top of the Maple tool bar (see Figure 6.7.2.1 below). This provides a graphical user interface to control animation playback. With the controls, one can, among other things,

1. Start playing the animation.
2. Stop the animation.
3. Display only a particular frame, "frozen".
4. Control the number of frames per second it plays.
5. Set it to play once or continually repeat in a loop.

The controls are highly similar to video playback controls such as found in on popular internet sites (e.g. You Tube), so we won't discuss them at length here. See Graphics->Animation->Animation Toolbar under the Table of Contents of the on-line Maple help.

Right-clicking on the animation will also produce a menu of operations that provide an alternative for controlling the animation.

8.9 Another example of animation

Problem

A boy throws a ball straight up in the air with an initial velocity of 15 miles per hour. Once released, the ball's position is described by the function \( x(t) = v_0 t - \frac{1}{2} g t^2 \), where \( v_0 \) is the initial velocity, and \( g \) is the force of gravity, \( g = \frac{32 \text{ feet}}{\text{sec}^2} \).

(a) How many seconds does the ball stay in the air? (b) Generate an animation that shows the ball's motion. \( t \) should be measured in seconds, and position in feet from ground level. Assume that the ball starts at 0 feet altitude even though this would be a bit unrealistic for a boy to do unless he was standing in a pit!

(c) Use the animation to determine roughly when the maximum altitude is, and what that altitude is.

(d) Find how fast the initial velocity should be so that the ball goes up over 30 feet.

Solution

First convert 15 miles per hour into a velocity in feet per second.

\[
    v_0 := \text{convert}(15, \text{units}, \frac{\text{miles}}{\text{hour}}, \frac{\text{feet}}{\text{second}})
\]
Define the gravitational constant

\[ g := -32 \]  

Next, define the function for position.

\[ x := (t) \rightarrow v_0 \cdot t + \frac{g \cdot t^2}{2} \]

Doing a rough plot of position versus time shows us roughly when the ball will hit the ground. We guess that it might take three seconds.

\[ \text{plot}(x(t), t = 0..3) \]

Oh, that's too much. The answer seems to be about 1.4 seconds, but we can use \textit{solve} to come up with an exact value.

\[ \text{solve}(0 = x(t), t) \]

\[ 0, \frac{11}{8} \]  

There are actually two solutions -- the obvious one is when \( t=0 \) and the ball hasn't yet been thrown.

To get the larger one, we compose the \textit{max} function with \textit{solve}.

\[ \text{flightTime} := \text{max}(\text{solve}(x(t) = 0, t)) \]

\[ \frac{11}{8} \]  

This answers part (a) of the problem.

We verify our computation by evaluating \( x \) at that time and finding that the position of the ball really is at altitude 0.

\[ x(\text{flightTime}) \]
Now we need to make the movie. We need to create a function which for time $t$, draws a point at the coordinate $(0, x(t))$. To illustrate what we mean, let's compute the position of the ball at $t=1$ second.

$$x(1.0)$$

We want the ball to be at position $(0, 6)$. A plot command that would do this would be:

```plaintext
plot([0], [6], style = point, color = red, symbol = circle, symbolsize = 30)
```

Similarly, at $t=.5$, the ball's position would be at $x(0.5)$:

```plaintext
plot([0], [x(0.5)], style = point, color = red, symbol = circle, symbolsize = 30)
```

Evidently $x(0.5)$ is 7.

We create a user-defined function that creates a plot structure as a result.

```plaintext
ballFrame := (t) -> plot([0], [x(t)], style = point, color = red, symbol = circle, symbolsize = 30)
```
Let's try this out for \( t = 0.5 \) and see if we get the same result.

\[
\text{ballFrame}(0.5)
\]

We can now use this function with \texttt{animate}. Note that we needed \texttt{flightTime} in order to describe how long the movie runs. If we did more or less than that, then the ball wouldn't have landed, or would be shown as going below ground level.

\[
\text{with}(\text{plots}) : \\
\text{animate(ballFrame, [t], t = 0 .. flightTime)}
\]

This answers part (b) of the problem.

By playing the movie, we see that the maximum altitude is about 7.5 feet, at time \( t = 0.687 \) seconds.

This answers part (c) of the problem.
To answer part (d), we need to do more programming. We first modify the plot so that it draws a line segment at 30 feet as well as plotting the position of the ball. We use the line function of the plottools package discussed in an ?? to draw a line segment at (-5,30) to (5,30), and to color it green:

\[ \text{height} := 30 \]

\[ \text{with(plottools)} : \]

\[ \text{pLine} := \text{line}([-5, \text{height}], [5, \text{height}], \text{color} = \text{"green"}) \]

\[ \text{CURVES}([-5,30.], [5., 30.], \text{COLOUR(RGB, 0., 1.00000000, 0.)}) \] (8.33)

The display function can be used to combine this line with a frame of the movie. Here is an example of this:

\[ \text{with(plots)} : \]

\[ \text{display([ballFrame(0.5), pLine])} \]

\[ \begin{array}{c}
\text{30} \\
\text{25} \\
\text{20} \\
\text{15} \\
\text{10} \\
\text{0} \\
\end{array} \]

The automatic scaling of plot chops off the vertical distance between 6 feet and 0 because there is nothing in this frame that needs that. In the animation, the scale is adjusted so that all frames operate in the same axes.

Now we can create a new user-defined function that plots both the line and the ball.

\[ \text{ballWithLine} := (t) \rightarrow \text{display([ballFrame}(t), \text{pLine})] \]

\[ t \rightarrow \text{plots:-display([ballFrame}(t), \text{pLine})] \] (8.34)

To look at the behavior of the ball at a particular velocity, we can now execute a two line script, consisting of assigning \( v0 \) to the desired initial velocity, and then the operation that draws the movie.

\[ v0 := \text{convert} \left( 15, \text{units}, \frac{\text{miles}}{\text{hour}}, \frac{\text{feet}}{\text{second}} \right) \]

\[ 22 \] (8.35)

\[ \text{animate(ballWithLine, [t], t = 0..\text{max}(...))} \]
As we already have seen, \( v\theta = 22 \) is not fast enough. We set it higher and recalculate the movie:

\[ v\theta := 50 \]

\[
\begin{align*}
\text{animate}(\text{ballWithLine}, [t], t = 0 .. \text{max}(\text{solve}(x(t) = 0, t)))
\end{align*}
\]

That was too high. Let's try 40.

\[ v\theta := 40 \]

\[
\begin{align*}
\text{animate}(\text{ballWithLine}, [t], t = 0 .. \text{max}(\text{solve}(x(t) = 0, t)))
\end{align*}
\]

Too low. Let's try 45.

\[ v\theta := 45 \]

\[
\begin{align*}
\text{animate}(\text{ballWithLine}, [t], t = 0 .. \text{max}(\text{solve}(x(t) = 0, t)))
\end{align*}
\]
So 45 feet per second seems to be about right. We could get a more precise determination through movie-watching, but for high accuracy we should use more mathematics. In a subsequent chapter, we will introduce additional Maple operations that can calculate the velocity exactly (or a close approximation) without the trial-and-error of movie watching. Having the movies did give us a better understanding of the phenomenon.

### 8.10 Exporting animations and non-animated plots

One operation available in the popup menu is Export. Right-click (or control-click) -> Export -> Graphics Interchange Format will produce an animation file in .gif format. As the animation file is being created, a dialog box will appear asking you to specify the directory where the .gif file should be written. Once created, the file can be included on web pages or other documents.

This feature is also available for ordinary (non-animated) plots. Right-clicking (control-click for Macintosh) will create a file of the plot in .gif, .jpeg, or .ps format. However, .gif file is the only format of the three that is supported by web browsers for animations.

**Exporting animations through the pop-up menu**
## 8.11 Summary of Chapter 8

### Using `with` to include external packages

<table>
<thead>
<tr>
<th>Package</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>plots</code></td>
<td>include the <code>animate</code> and <code>display</code> functions.</td>
</tr>
<tr>
<td><code>plottools</code></td>
<td>includes functions for drawing various shapes, such as <code>line</code> and <code>circle</code>.</td>
</tr>
<tr>
<td><code>CurveFitting</code></td>
<td>includes the <code>LeastSquares</code> function.</td>
</tr>
</tbody>
</table>

### The `LeastSquares` function

The `LeastSquares` function produces a best-fit line using the least squares technique. Input two sets of data and the function will produce the equation of the best-fit line.

```plaintext
pData := [134.2, 142.5, 155.0, 159.8, 171.1, 184.2];
[134.2, 142.5, 155.0, 159.8, 171.1, 184.2]  

tData := [0, 20.1, 39.8, 60.0, 79.9, 100.3];
[0, 20.1, 39.8, 60.0, 79.9, 100.3]  

pressureFormula := LeastSquares(tData, pData, t)  
133.500490 + 0.4858370741 \* t
```

### Combining plots and shapes using `display`

Supplying the `display` function with a list of two or more plots will cause those plots to be plotted on top of one another.

We can save a plot in a variable to display later.

Marking a specific value on the plot can be accomplished using the `line` or `circle` function.

```plaintext
display([plot1, plot2, plot3])

plot4 := plot(sin(x), x = 0..2*Pi)  
PLOT(....)

maxImp := line([0, 1], [2*Pi, 1], color = "blue")  
CURVES([[0, 1], [6.283185308, 1]], COLOUR(RGB, 0, 0, 1.00000000))

display([plot4, maxImp])
```
Combining plots and shapes using display

Parameterized plots
For plots that may have values corresponding to 2d positions, we use multiple functions to define both $x(t)$ and $y(t)$.

\[ x_{pos} := (t) \rightarrow 3\cos(t) \]
\[ y_{pos} := (t) \rightarrow 2\sin(t) \]

We put these functions in a list, along with the range of the independent variable $t$, and plot.

\[ \text{plot}([x_{pos}(t), y_{pos}(t), t=0..\pi], \text{scaling} = \text{constrained}): \]

Animating plots using animate
Creating an animation

\[ x := (t) \rightarrow 50 \cdot \sin\left(\frac{\pi}{4}\right) t : \]
\[ y := t \rightarrow 50 \cos\left(\frac{1\pi}{4}\right) t - 16 t^2 : \]
\[ \text{posPlot} := (t) \rightarrow \text{plot}([x(t)], [y(t)], \text{style} = \text{point}, \text{symbol} = \text{circle}) : \]
\[ \text{animate(posPlot, [t], t = 0..2)} : \]

Controlling the animation using the animation tool bar or the animation popup menu
The animation tool bar is located above the workspace window, while the popup menu can be displayed by right-clicking the animation.

Exporting an animation to a graphics file
Right click animation>Export>Graphics Interchange Format
9 Moving into programming

9.1 Chapter Overview

The interactive document interface that we've worked with so far is good for quick development of calculations involving a few steps. Because feedback occurs after each step entered, this style of working is often the fastest way of getting such short calculations done. We've also seen how the effort to develop a script can be made to yield greater payback by finding and exploiting situations that require re-use, by editing parameters of a script and re-executing it.

In our previous work, we've executed a script by the following process:

a) Setting up the values needed for the parameters assigned at the beginning of the script.

b) Positioning the mouse cursor on the first line of the script and hitting return (or enter) repeatedly, or Selecting the entire script with the mouse, and then executing it all with Edit->Execute->Selection. With longer scripts or extensive re-execution, this process becomes laborious even if we are saving a lot of time by having the computer do the work.

In this chapter, we introduce a new way of entering and executing scripts: code edit regions. While such regions make it more convenient to work and execute blocks of Maple instructions, the programmer must enter things using the textual version of Maple expressions and operations. With code edit regions, the programmer must also work harder to see how the state of the computation is changing during the computation, rather than only looking at the final output. This is because if the final result is wrong, the programmer must find at what point in the script mistakes (called program bugs) occur. Work with code edit regions often requires adding print or printf statements into the script to better see changes in variables and intermediate results.

Being able to work with longer scripts conveniently also allows us to solve more complicated problems. But keeping all the ideas in your head for what you want to do on the computer becomes too error-prone as you start dealing with more complicated tasks. The programmer takes on two kinds of work -- the plan for how to solve the problem, and the coding where the plan is communicated to the computer in a programming language. Despite Maple's powerful repertoire of functions (e.g. "solve" or "fit data"), it is often the case that the steps of a plan can require several Maple actions to implement. It is often productive to outline a solution before writing the details of the code. The outline (as well explanations of any tricky details of the code) can be included in the code edit region by including program comments. Furthermore, all the details of the plan need not be made in one fell swoop. A programmer can in comment form create a broad outline for a solution and then progressively refine it, similar to how an author uses outlining to create a paper or report.

Once a good outline has been developed, the programmer can proceed to develop the code incrementally, adding a line or two of code at a time, running the entire region and checking that it works as far as it goes, and then proceeding onto another few lines. The alternative to incremental development -- typing in the entire solution and then trying to fix its problems -- is much more time consuming because of the difficulties in finding the cause of problems when there are many possible culprits for a bug.

9.2 More on printing: print, printf and sprintf

print is a function that takes a sequence of values as arguments, and returns NULL (first introduced in ???) as a result. Its use is in its side effect -- it causes the sequence of values to be displayed in the worksheet in two dimensional format. print is used to provide more intelligible displays of information by providing words to go along with quantities or formulas that are computed. The words, of course, are provided by including strings as part of the sequence.

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use of print</td>
<td>We set up a list of values. L has all floating-point values because the &quot;10.0&quot; in the denominator causes all computations to be done using limited-precision arith-</td>
</tr>
</tbody>
</table>
Example

```plaintext
print("This list presents some values of the function ", exp(x));
print( ", The last element of the list is ", L[-1]);

"This list presents some values of the function ", e^x
". The last element of the list is ", 0.7408182207

print("There are ", nops(L),
 " elements in this list. The largest value is", max(L))

"There are ", 3, " elements in this list. The largest value is", 0.9048374180
```

Commentary

print prints out the sequence of values given to it as arguments. You can give any number of items in the sequence. The commas separating the string and the number are from the sequence. Note that as first mentioned ???, the result is not displayed because the line ends with a colon.

```plaintext
printf returns NULL (first introduced in ???) as its result, and is used primarily for its side effect of causing information to be displayed in the worksheet. The first item in the sequence is a string that contains ordinary words plus special format codes that begin with %, ??? summarizes the commonly used the format codes. The on-line documentation for printf describes all format codes comprehensively. Some of them allow for quite intricate effects such as right or left justification, padding with leading or trailing zeroes, etc.

Example

```
Use of printf

L := [ exp(-1), exp(-2), exp(-3), 4/5 ];

printf("The last element of the list is %f", L[-1])
The last element of the list is 0.800000.

printf("There are %d elements in this list. The largest value is", nops(L),
 max(L))
There are 4 elements in this list. The largest is 9.048374e-01.

printf("L has %d elements. nops(L) ", nops(L)); printf("The largest is %e", 
max(L));
L has 4 elements. The largest is 9.048374e-01.

printf("There are %d elements in this list. The largest is %e", nops(L),
 max(L))
There are 4.000000 elements in this list. The largest is
Error, (in fprintf) integer expected for integer format

printf("%d", L)
Error, (in fprintf) number expected for floating point format

printf("%a", L)
[.9048374180, .8187307531, .7408182207, 4/5]
```

Commentary

We set up a list of values as in the previous example.

The first argument to printf is a string describing how to format the rest of the values. The string contains format codes describing where to insert the values of the later arguments into the message described in the string. In this example, there is one format code %f, for printing numbers in conventional decimal point notation. Since printf joins together the message and the value together into a single message without a sequence into a single result, what is printed is without commas.

Additional printf codes are "%d" (for integer values), and "%e" (for floating point numbers in scientific notation). The format code \n causes the next output to occur on the next line.

Note that a printf statement does not automatically cause the output on a new line. Thus, several printf statements executed together as these are will have all the output on a single line.

If you attempt to use %f format on an integer, or %d format on a number that is not an integer, you will get incorrect values (with no warning that they are incorrect), or possibly an error message. Let the user beware; the computer isn't going to necessary save you from making a mistake.

You can't print a list of floating point numbers using just %f format.

The %a format will handle arbitrary numerical or non-numeric values. The value being printed out is a list, which is a "non-numeric value" be-
sprintf is like printf in that its first argument is a string with format codes, and the rest of its arguments are values to be formatted. However, unlike printf it does not print (display) any information, it returns a string which is the formatted information instead. This is useful for users that need to take the formatted information and use them in other ways.

**Use of sprintf**

We set up a list of data points. Using sprintf, we create a string. The %d format is used to insert the number of points into the message.

```
L := [exp(-1), exp(-2), exp(-1.5), exp(-2.7)]:
message := sprintf("plot of x versus exp(-x), using %d points", nops(L))
```

We need the string to put a title onto our pointplot. You can read more about the `title=` argument to pointplot by entering `plot,options` in Maple's on-line help.

```
plot([-1,-2,-1.5,-2.7],L,style=point,title=message)
```

**Common formatting codes for printf and sprintf**

<table>
<thead>
<tr>
<th>Format code</th>
<th>Example</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td><code>printf(&quot;The answer is: \%a&quot;, x=x^2+1)</code></td>
<td>Algebraic format. Works for any type of value. The value is printed out in a format suitable for textual input.</td>
</tr>
<tr>
<td>%f</td>
<td><code>printf(&quot;The answer is: \%f&quot;, L[-1])</code></td>
<td>Fixed format for floating point numbers. w and d are optional. If given, w describes the number of columns for the entire number, and d the number of columns for the number after the decimal point.</td>
</tr>
<tr>
<td>%w.df</td>
<td><code>printf(&quot;\%w.\%df&quot;, L[-1])</code></td>
<td>Values will be converted to floats where possible.</td>
</tr>
<tr>
<td>Format code</td>
<td>Example</td>
<td>Explanation</td>
</tr>
<tr>
<td>-------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>printf(&quot;%d&quot;, 3/4)</code></td>
<td>0.750000</td>
<td>(9.14)</td>
</tr>
<tr>
<td><code>printf(&quot;The answer is. %d&quot;, x=x^2 + 1)</code></td>
<td>The answer is:</td>
<td></td>
</tr>
<tr>
<td><code>printf(&quot;Error, (in fprintf) number expected for floating point format&quot;)</code></td>
<td>Error, (in fprintf) number expected for floating point format</td>
<td></td>
</tr>
<tr>
<td><code>%e</code></td>
<td><code>printf(&quot;%e&quot;, L[-1])</code></td>
<td>(9.16)</td>
</tr>
<tr>
<td></td>
<td><code>9.095277e-03</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>printf(&quot;%12.5e&quot;, L[-1])</code></td>
<td>(9.17)</td>
</tr>
<tr>
<td></td>
<td><code>9.09528e-03</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>printf(&quot;%e&quot;, 10!)</code></td>
<td>(9.18)</td>
</tr>
<tr>
<td></td>
<td><code>3.628800e+06</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>printf(&quot;%e&quot;, 3/4)</code></td>
<td>(9.19)</td>
</tr>
<tr>
<td></td>
<td><code>7.500000e-01</code></td>
<td>(9.20)</td>
</tr>
<tr>
<td></td>
<td><code>printf(&quot;%10.5e&quot;, 20!)</code></td>
<td>(9.21)</td>
</tr>
<tr>
<td></td>
<td><code>2.43290e+18</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>printf(&quot;%e&quot;, 3/4)</code></td>
<td>(9.22)</td>
</tr>
<tr>
<td></td>
<td><code>7.500000e-01</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>printf(&quot;The answer is. %e&quot;, x=x^2 + 1)</code></td>
<td>(9.23)</td>
</tr>
<tr>
<td></td>
<td>The answer is:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Error, (in fprintf) number expected for floating point format</td>
<td></td>
</tr>
<tr>
<td><code>%d</code></td>
<td><code>printf(&quot;%d&quot;, 20!)</code></td>
<td>(9.24)</td>
</tr>
<tr>
<td></td>
<td><code>2432902008176640000</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>printf(&quot;%d&quot;, 3/4)</code></td>
<td>Error, (in fprintf) integer expected for integer format</td>
</tr>
<tr>
<td></td>
<td><code>printf(&quot;%d&quot;, 3.0)</code></td>
<td>Error, (in fprintf) integer expected for integer format</td>
</tr>
<tr>
<td></td>
<td><code>whattype(3.0)</code></td>
<td><code>float</code></td>
</tr>
<tr>
<td></td>
<td><code>printf(&quot;%d&quot;, 3)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>3</code></td>
<td>(9.25)</td>
</tr>
<tr>
<td></td>
<td><code>whattype(3)</code></td>
<td></td>
</tr>
</tbody>
</table>

Scientific format for floating point numbers. wand d are optional. If given, w describes the number of columns for the entire number, and d the number of columns for the number after the decimal point. If w is wider than the number of digits required, then it is filled out with leading white space. Values will be converted to floats where possible.

Format for integers. This format rejects any value that is not of integer type. The built-in whattype function can provide guidance about what is and what isn't going to work, if you aren't sure.
<table>
<thead>
<tr>
<th>Format code</th>
<th>Example</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>%s</td>
<td>`c := &quot;DodgerBlue&quot; ;</td>
<td>Format for strings.</td>
</tr>
<tr>
<td></td>
<td><code>printf(&quot;The color is %s.&quot;, c);</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>The color is DodgerBlue.</td>
<td></td>
</tr>
<tr>
<td>\n</td>
<td><code>printf(&quot;One two,\nbuckle my shoe\n%d,%d,\nshut the door &quot;. , 3, 4)</code></td>
<td>Causes a line break between preceding and succeeding parts of the output.</td>
</tr>
<tr>
<td></td>
<td>One two, buckle my shoe. 3, 4, shut the door.</td>
<td></td>
</tr>
<tr>
<td>\t</td>
<td><code>printf(&quot;%d,\t%d,\tbuckle my shoe\n%d\t%d,\tshut the door &quot;. , 1, 2, 3, 4)</code></td>
<td>Causes next output to appear starting at the next columnar (tab) position.</td>
</tr>
<tr>
<td></td>
<td>1, 2, buckle my shoe. 3 4, shut the door.</td>
<td></td>
</tr>
</tbody>
</table>

### 9.3 For the curious: why printf is so cryptic

The format codes for `printf` are a "programming language within a programming language" in that they describe in extremely abbreviated form how the computer should output values. The style used for format codes is very much unlike the rest of Maple, in that brevity rather than readability seems to be the criterion used to design this portion of the language. A language for science and engineering popular starting in the middle of the twentieth century was Fortran, which had formatting that was similar in approach but was much different in syntax. Given your knowledge of Maple (and English), the intent of the program is fairly obvious: it will print There were 3 values, and the answer was 4.7.

A Fortran program that demonstrates formatted printing

```fortran
n = 3
x = 4.7
printf(*,35) n,x
35 format('There were',i2,' values, and the answer was',f5.2,' ')
stop
end
```

The design of `printf` can be explained by the fact that Maple's `printf` is a close imitation of the `printf` found in the programming language C (popular with many programmers at the time Maple was invented in the '80s, and still popular among computer engineers and scientists doing device programming). Matlab, being invented in the same era, also has a `printf` function of highly similar design. Imitation makes it easier to transfer programming skill from one language to another. This leads us to observe that the inventors of programming languages will tend to imitate unless they have a goal or insight that justifies a difference. Programming languages designed by skillful designers do not introduce arbitrary differences without a good cause. When you notice similarities between languages, take it as a recognition that many people liked that style of doing things, across many different kinds of programmers and kinds of problems. When you notice a difference between languages, you should explore the reasons why the designers made different choices. Usually there is a problem or issue that the designer is trying to handle better by being different from the rest of the pack. This knowledge leads to greater appreciation of the way a language does things.
9.4 Code edit regions: executing a series of actions at once

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.

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 for each statement will appear below the region in blue, except for the statements whose printing is suppressed by a colon.

Any portion of a line that begins with a "#" is a program comment. The rest of the line is regarded by Maple as something that people will read, not an instruction to be performed by the computer. Typically what appears after the # is commentary written in English (or whatever language is convenient for communication with the intended audience) that helps explain/remind human readers what a segment of code is about. As you read programs with comments, you will see that some programmers are "chattier" than others. This is because they feel that their intended reading audience needs more explanation.

In professionally written code, you will often see comments at the beginning of the region that give an overview of what the code region does, the name of the author(s), and the date of creation/modification. Professional organizations
often mandate a particular style and content for such "header" comments, so that it will be easy to find information about any code written by several programmers working on a single project.

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

<table>
<thead>
<tr>
<th>Code region</th>
<th>Commentary</th>
</tr>
</thead>
</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.  | Segments of lines that begin with # are regarded as program comments (for the program reader's eyes), not operations for Maple to carry out. |
|  
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. |

The code region will develop a scroll bar 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.

Enlarging a code Region window by changing component properties via the clickable menu
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. Clicking on the code icon will execute the code within.

### A collapsed code edit region

<table>
<thead>
<tr>
<th>Code region</th>
</tr>
</thead>
</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. |

<table>
<thead>
<tr>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>We collapsed the region by positioning the cursor in the code region window, and then entering right click-&gt; Collapse Code Edit Region. Note that the comment on the first line becomes displayed next to the code edit icon.</td>
</tr>
</tbody>
</table>
9.5 Turning intentions into code

More complicated problems or longer computations requires a planning phase before any programming is done. Once the plan has been developed, the programmer then goes about the business of writing the actions described in the plan in the programming language being used. This business of translating intentions into instructions in a language a computer can understand is called **coding, or writing program code**. In concrete terms for this course, what this means is that the problems we will be working on will need some planning before anything is written in Maple. As with all plans, it is beneficial to be able to see how well the plan is going to work (i.e. to evaluate the plan) and fix/polish it before moving to the coding phase.

The purpose of programming languages is to express a solution to the computer. They are not ideal languages for people to think in to develop the outline of a solution.

9.6 Making code easier to read and understand

As has been mentioned ???, an important part of training for a programmer is to make code easier to reuse, since reuse allows the cost of writing the program to be amortized over more invocations of it. Part of reuse is allowing others to understand what they should do with the code, or to allow yourself to more quickly recall the details of what you were doing when you return to a program after a few weeks or months away from it. This kind of reuse is helped by making certain stylistic choices in how you write the code. Here are some ideas:

1. Use mnemonic variable names -- names that suggest what the variable's purpose is. This is different from the variable naming practices of the "mathematics culture", which is to use symbols of one letter in length, e.g. \(x, i, \alpha\). The reason for the difference in practices is that it is relatively easy to keep track of the purpose of variables if the expression is only one line long, as is typical in elementary mathematics. Even elementary programs however consist of many lines. A first-year student's program might be a dozen or even 100 lines, using ten or twenty variables. Remembering what they all do becomes too hard without reminders.

2. Invent and use user-defined functions to encapsulate common operations. Reusing a name two or more times has advantages beyond not doing so much typing. It allows the reader to see quickly that you are doing the "same thing as before" instead of something subtly or radically different.

3. Mnemonically name functions. If the function has a name that describes what is doing (e.g. "plotCoolingFunction" instead of "f") then it becomes even easier for the reader to understand/recall what the function is intended to do even the first time that they read about it.

4. 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 variables 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.

5. Indentation is used for indicating nuances of control, such as the extent of code blocks that are repeated. We will discuss indentation more in the next chapter where the kind of code that benefits from it is first discussed.
9.7 Troubleshooting common errors in entry

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. Trying to enter too many lines at once before testing what you have so far can make older errors appear more mysterious.

Example

<table>
<thead>
<tr>
<th>Code regions with entry errors</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Example</strong></td>
<td></td>
</tr>
<tr>
<td><code>a := sin(3.5 + cos( Pi/2) ;</code></td>
<td>The error message indicates that a semi-colon was not expected where Maple scanned it. This is typically semi-colons can appear only at the end of a complete Maple statement. This message typically indicates that something went wrong in what was entered before the semi-colon. Edit the code region to make the statement correct, and re-execute the region.</td>
</tr>
<tr>
<td><strong>Error, <code>;</code> unexpected</strong></td>
<td></td>
</tr>
<tr>
<td>(Above region with fixes)</td>
<td></td>
</tr>
<tr>
<td><code>a := sin(3.5) + cos( Pi/2) ;</code></td>
<td>The problem here is a missing parentheses ). To fix this, you'd put a closing ) after the &quot;3.5&quot;, and then re-execute the region with a control-e (or command-e).</td>
</tr>
<tr>
<td>-0.3507832277</td>
<td></td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td></td>
</tr>
<tr>
<td><code>a := sin(3.5) + cos( Pi/2)</code></td>
<td>This is only a warning, and executing this region does perform the computation correctly.</td>
</tr>
<tr>
<td><strong>Warning, inserted missing semicolon at end of statement</strong></td>
<td></td>
</tr>
<tr>
<td>-0.3507832277</td>
<td></td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td></td>
</tr>
<tr>
<td><code>a := sin(3.5) + cos( Pi/2)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>b := cos(3.5) + sin( Pi/2)</code></td>
<td></td>
</tr>
<tr>
<td><strong>Error, missing operator or <code>;</code></strong></td>
<td></td>
</tr>
<tr>
<td>(Above region edited with fixes)</td>
<td></td>
</tr>
<tr>
<td><code>a := sin(3.5) + cos( Pi/2) ;</code></td>
<td></td>
</tr>
<tr>
<td><code>b := cos(3.5) + sin( Pi/2);</code></td>
<td></td>
</tr>
<tr>
<td>-0.3507832277</td>
<td></td>
</tr>
<tr>
<td>0.0635433127</td>
<td></td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td></td>
</tr>
<tr>
<td><code>a := sin(3.5 + cos( Pi/2)) ;</code></td>
<td>If you try to enter several lines at once before entering, you have the additional problem of figuring out which line the mistake occurred on. In this case, you have to decide which of the two semi-colons was unexpected.</td>
</tr>
<tr>
<td><code>b := cos(3.5) + sin( Pi/2);</code></td>
<td>Fortunately, you can sometimes get some help if you are observant -- the cursor will often flash on the line where the problem is first noticed (the first line in this case).</td>
</tr>
<tr>
<td><strong>Error, <code>;</code> unexpected</strong></td>
<td></td>
</tr>
<tr>
<td>(Above region edited with fixes)</td>
<td></td>
</tr>
<tr>
<td><code>a := sin(3.5) + cos( Pi/2)</code></td>
<td>There's both a missing closing parentheses and a missing semi-colon here. The symptom is that there is a warning message (which isn't necessarily a sign of trouble, but is here), plus the fact that you are expecting a value to be printed from the computation but just see a red &quot;&gt;&quot;, with no numeric output.</td>
</tr>
<tr>
<td><strong>Warning, premature end of input, use &lt;Shift&gt; + &lt;Enter&gt; to avoid this message.</strong></td>
<td>Fixing both problems and re-executing the region will produce the correct result.</td>
</tr>
<tr>
<td><code>a := sin(3.5 + cos( Pi/2)</code></td>
<td>Sometimes when you enter a line, you make more than one error. Even though the same mistake has been made as in the previous</td>
</tr>
<tr>
<td><code>b := cos(3.5) + sin( Pi/2);</code></td>
<td></td>
</tr>
</tbody>
</table>
example, there is a different error message than before. Executing this region causes the message indicated, plus the cursor will be observed to flash at the start of the second line.

Rather than being in an "I was expecting more" state which produced the first warning message, Maple is in a "I was expecting more, but wait what I've seen isn't what I was expecting". Since there's nothing before the cursor started flashing before any of the second line was parsed, it means that the error must have happened before that on the previous line.

There are still two things to fix. Fixing them will produce two lines of output.

### 9.8 The "outline approach" for developing a solution in a code region.

Many students learn to write a report through the "outline method". See for example, [http://owl.english.purdue.edu/owl/resource/544/02/](http://owl.english.purdue.edu/owl/resource/544/02/). The outline method suggests that the way to proceed goes roughly like this:

1. After considering what you would like to say, create the major sections for your content, and put them into an order that makes sense.

2. Refine each section, by adding subsections that group the major points of each section. Put the subsections in an order that will make sense to the reader.

3. Continue in this way to create subsubsections, etc. until the level of detail is small enough that you can see an easy writing task for each segment in your outline.

The final version of the report will keep the content organized in this way. This kind of refinement can make it easier to keep things well-organized and coherent at each level of detail. It also helps avoid getting lost in the fine details.

Code development using this style has the initial outline written as program comments. Only when the subpoints or subsubpoints can be expressed straightforwardly as something the programmer finds easy to code (or one line of code) does the writing turn from writing comments into writing of code.

This makes commenting something you do even before you are completely certain as to what you will do. It's a lot easier to change comments than it is to redo the details of code that has already been written. Having the plan description included into the code region means that you do not have to jump between pieces of paper and the Maple worksheet to check whether the plan is being followed by the code.

### A problem to solve

(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°F 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°F to 120°F 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°F to 130°F? How long will the coffee temperature remain between 110°F and 130°F.

Using the method we suggest, we would first create a code region that has only comments, giving the basic outline of what we would do. In this situation, it would also be good to create a preface describing the problem. In Maple, we could just place the description as ordinary text before the code region with the comments.
We think about what we would do and realize that these questions can be answered by doing the "heating and cooling equation" calculations we've seen. Such equations typically require finding a "heating constant" used in an exponential formula, and then solving an one or more equations for t.

Initial code outline to solve the problem

<table>
<thead>
<tr>
<th>Description of problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>(From Sullivan, Pre-calculus, p. 342)</td>
</tr>
</tbody>
</table>

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?

### Define shared parameters for problem

### Define parameters for each company

### Answer the questions for CentiKeeper

### Answer the questions for Temp Control

### Answer the questions for Hot'n'Cold

Each of these steps needs to be fleshed out a bit more before we can start writing lines of code. We don't have to flesh out things starting from the beginning, we can add more details to the outline in any appealing order. In this case, we talk about how to answer the questions for CentiKeeper, using the solution process discussed in the outline approach as a model for the computation. We still have only comments, not code. We can use two ##s at the start of the line (which, since it begins with a #, is still regarded as a comment) to indicate that the comment is at a lower level than before.

Next step of refinement for code outline

<table>
<thead>
<tr>
<th>#Define shared parameters for problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Define parameters for each company</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#Answer the questions for CentiKeeper</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Find the heating constant k1 for CentiKeeper</td>
</tr>
<tr>
<td>#Substitute the constant into the heating equation.</td>
</tr>
</tbody>
</table>

| #Find the value of t such that the heat equation produces 130 degrees. |
| #Find the value of t such that the heat equation produces 110 degrees. |
The general heating formula we're going to use is \( u(t) = T + (u_0 - T)e^{kt} \). \( u_0 \) is the initial temperature, and \( T \) is "room temperature". \( k \) is the heating constant. We realize that there are other entities too which we give mathematical names to: \( \tau \), a period of time elapsed after the start, and \( N \), the temperature of the liquid in the cup after \( \tau \) minutes has elapsed. We will use different \( u_0, T, \tau, N \) and \( k \) for each company. We will give different names to the programming variables that we will use for each company, rather than trying to re-use the symbols several times.

We can flesh out the outline further with these decisions.

**Next step of refinement for code outline**

```plaintext
# Define shared parameters for problem
## There are no shared parameters

# Define parameters for each company
## u01, T1, k1, tau1, N1 for CentiKeeper
## u02, T2, k2, tau2, N2 for Temp Control
## u03, T3, k3, tau3, N3 for Hot'n'Cold

# Answer the questions for CentiKeeper
## Find the heating constant k1 for CentiKeeper
## Substitute the constant into the heating equation.

## Find the value of t such that the heat equation produces 130 degrees.

## Find the value of t such that the heat equation produces 110 degrees.

## Compute the difference, print out the answer.

# Answer the questions for Temp Control
## Do the same thing as we did for CentiKeeper, only use the variables we have for Temp Control

# Answer the questions for Hot'n'Cold
## Do the same thing as we did for CentiKeeper, only use the variables we have for Hot'n'Cold
```

We still haven't written any code, but we are about ready to do that. First however, we should learn a few more "tricks" to efficient code development.
9.9 Debugging's most crucial skill: identify the first line of code where things go wrong

The execution trace of a code region is the output that results from executing each line of a block of code in a region. Full execution traces can be created only after the syntax errors have been removed from the code region so that the region can be executed without any error messages. Removing error messages is only the beginning to the work, however. The code you write may still have mistakes in it -- the instructions you give may make grammatical sense, but be the wrong actions to solve the problem. A mistake in code is often called a bug, and removing mistakes from code is called debugging it. Programmers spend a lot of time debugging, usually because getting something new to work is difficult. Any ideas, approaches, or strategies for making debugging less difficult are usually welcomed by programmers.

Using an execution code trace to find a bug in a short code block

```plaintext
#Define parameters for each company
##u01, T1, k1 , tau1, N1 for CentiKeeper

u01 := 200;
T1 = 70;
N1 := 100;
tau1 := 30;

200
T1 = 70
100
30
```

After creating the code region and entering the lines, we get the execution trace -- the lines in blue -- by clicking on the code edit region and typing either control-e (command-e for Macintosh), or entering right-click->Execute Code. Each line in blue corresponds to result of execution of a line of code.

A major difference between executing a script interactively in a document, and in a code edit region, is that execution trace for interactive execution occurs intermingled with the lines of code, while with code regions the execution trace occurs after the listing of all the lines of the block.

Programmers debugging code inspect execution traces to mistakes in their programming. They try to find the first line of code where things go wrong. While the cause of that "buggy result" may lie in something that happened even earlier, it is certain that the first program bug did not happen later. Knowing the first buggy line of the execution trace typically reduces the number of lines of code that must be inspected for the mistake, since the error must have occurred in the code at or before the line that generated the "bad output".

In the execution trace above, we see that the first line of output is consistent with the "u01 := 200;" line of code -- the value of the assigned variable is printed out. We can see immediately that the T1=70 line looks different than the others, which were just numbers from the assignments. Looking at the code, we see that there is a difference between the second line of code (excluding comments) and the others -- there is an "=" rather than a ":=". We have used the execution trace to discover a mistake we have made in a line of code -- that line should really be T1 := 70.
9.10 Using print and printf to elaborate the execution trace

Sometimes the execution trace does not provide enough information to discover a bug. For that reason, programmers sometimes rely on placing `print` or `printf` statements in code regions just before steps that they want to look at. Usually the statements that do the printing output a message that includes not just values of variables, but what variables they are, and other information that makes it easier to identify which line of code out of many in the code block is doing the printing.

Continuing the example above, we enter more lines of code and begin to execute what we have. We ran into problems where some of the variables got mangled from bugs in previous executions of the code region. So we place a "restart" at the beginning to make sure that we unassign all variables before we begin execution again, leaving only the assignments made this time around.

We put in a printf statement to make it clear what we're solving for, before we solve it. We do this because we know (from previous efforts working with this kind of problem) that solving this "heat equation" can be tricky.

### Execution trace augmented by a printf

```plaintext
#define parameters for each company
## u01, T1, k1, tau1, N1 for CentiKeeper
reinitialize;  #Unassign variables from previous executions
u01 := 200;
T1 := 70;
N1 := 100;
tau1 := 30;

#Answer the questions for CentiKeeper
## Find the heating constant k1 for CentiKeeper
### Describe the heat equation
heat1 := T1+(u01-T1)*exp(k1*t);
ht1 := eval(heat1, t=tau1);
printf("Solving for %a",ht1);
k1soln := solve(ht1,k1);
printf("Solving %a for heat constant: k1 is %f", ht1, k1soln);

200
70
100
30
70 + 130e30k1
70 + 130e30k1

Solving for 70+130*exp(30*k1)

\[ \frac{1}{30} \ln \left( \frac{7}{15} \right) + \frac{1}{30} \pi \]

Solving 70+130*exp(30*k1) for heat constant: k1 is

Error, (in fprintf) number expected for floating point format
```
We see a lot of gibberish after awhile, but our attention turns to the "Solving for 70+130*exp(30*k1)" and the next line of the execution trace, which we infer from its placement is the result assigned to k1soln. It appears to be an imaginary number (because it has an "I" in it), which is definitely not the right kind of value for a cooling constant. Looking again at the printf output, we see that what we're solving for is wrong -- according to the math we should be solving the equation $100 = 70 + 130e^{30k1}$. We use these insights (which come only after we understand what we would be doing by hand with this calculation), to rework the code region so that it sets up an equation that uses the "N1" information.

### Execution trace from a repaired code region

```plaintext
# Define parameters for each company
## u01, T1, k1, tau1, N1 for CentiKeeper
restart;  # Unassign variables from previous executions
u01 := 200;
T1 := 70;
N1 := 100;
tau1 := 30;
## u02, T2, k2, tau2, N2 for

# Answer the questions for CentiKeeper
## Find the heating constant k1 for CentiKeeper

## Describe the heat equation
heat1 := N1 = T1 + (u01-T1)*exp(k1*t);
ht1 := eval(heat1,t=tau1);
printf("Solving for %a.",ht1);
k1soln := solve(ht1,k1);
printf("Solving %a for heat constant: k1 is %f.", ht1, k1soln);

200
70
100
30
100 = 70 + 130e^{30t}
100 = 70 + 130e^{30k1}

Solving for 100 = 70+130*exp(30*k1).

\[ \frac{1}{30} \ln \left( \frac{3}{13} \right) \]

Solving 100 = 70+130*exp(30*k1) for heat constant: k1 is -0.048878.
```

This presents information in the trace that convinces us that the right thing is going on.
9.11 Incremental coding and testing

The old adage "don't bite off more than you can chew" also applies to coding. Suppose you enter thirty lines of code and then execute them. If you find a mistake at the end, there are thirty lines that could be the source of the problem. Having to look through thirty of anything (the execution trace, or the code) can be time consuming and complicated to keep track of. You may have noticed that we have developed code in the past few sections only a few lines at a time -- entering them, viewing the execution trace, finding things to fix, and proceeding. It would be nice if we could just get everything written correctly the first time, but it is more realistic to assume that you won't. There will be typos and misconceptions to fix all along the way.

Proceeding incrementally makes it likely that it's always the last lines you entered where the problem is most likely to lie. Combining this with information from print statements to augment the execution trace can be a very efficient way of developing code because the execution trace information is presented intelligibly, and its typically only the last few lines that you need to scrutinize carefully.

Finally, you can use the "comment trick" to handle things incrementally even if you've already written more than two lines of code. You can type in all the lines of code, but add a # in front of all the lines except the first one. If you then execute the code region, you will see the results from only the first line of code because all the others will be treated as comments and not executed. Once you have that line working well, then you can remove the # from the second line, and re-execute the region. Because two lines are now not comments, you will see the execution trace from two lines of code. You can continue "uncommenting" lines in this fashion to achieve incremental development.

The comment trick can be used to on print or printf statements that you have been using for troubleshooting but are not completely sure that you no longer need. By putting a # in front of a print statement, you can turn off its effects without erasing it completely. If you decide that you need to see it again, you can remove the # and re-activate it. This can save a lot of typing.

Having proceeded to solve one version of the problem partially, let's see if we can create the whole ball of wax for CentiKeeper. We enter one new line at a time, execute the region, and see what needs fixing. After about five minutes, we have all the lines working, and have printed the desired answer at the end. Note that we have used an advanced feature of so that it prints only two digits after the decimal point. We also rely on the fact that even if the value being given to a %f format code isn't a floating point number, the result is automatically approximated so that it can appear in floating point format. You can read about these feature in the on-line help for printf.

At this point, we'll clone the process for the other two situations through copying, pasting, and editing to use different variable names.

**Execution trace augmented by a printf**

```plaintext
#define parameters for each company
##u01, T1, k1, tau1, N1 for CentiKeeper
restart;  #Unassign variables from previous executions
u01 := 200;
T1 := 70;
N1 := 100;
tau1 := 30;

#define the questions for CentiKeeper
##Find the heating constant k1 for CentiKeeper

###Describe the heat equation
heat1 := T1+(u01-T1)*exp(k1*t);
ht1 := N1=eval(heat1,t=tau1);
printf("Solving for %a.,ht1);
```
\[
\text{Solving for } 100 = 70+130e^{30k1}.
\]

\[
70 + 130e^{30k1} = 100
\]

\[
\ln \left( \frac{3}{13} \right)
\]

\[
\frac{1}{30} \ln \left( \frac{3}{13} \right)
\]

\[
\text{Solving } 100 = 70+130e^{30k1} \text{ for heat constant: } k1 \text{ is } -0.048878.
\]

\[
70 + 130e^{\frac{1}{30} \ln \left( \frac{3}{13} \right)}i
\]

\[
\frac{30 \ln \left( \frac{6}{13} \right)}{\ln \left( \frac{3}{13} \right)}
\]

\[
\frac{30 \ln \left( \frac{4}{13} \right)}{\ln \left( \frac{3}{13} \right)}
\]

The CentiKeeper cup keeps coffee warm for 8.30 minutes.
9.12 Using interactive development first for a tricky line of code

We've already mentioned an idea to speed up development of code that you're not completely sure about -- copy examples from documentation and get them to work in a worksheet, then modify the working example to do what you really would like to do. When developing a sizable script in a code region, there will still be spots where you won't know what to enter immediately. When you encounter such a situation, open a fresh Maple document (i.e. by doing File->New->Document Mode) and use it as an experimental sandbox working interactively to get those lines of code to work. Then copy the working code into the code region when it is "fully baked" or nearly so. If you have developed the code interactively using 2D math notation, you may need to convert it into ordinary textual format by applying Format->Convert To->1D Math Input.

### Converting 2D expressions into 1D expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x(t) = \frac{m v0 x (1 + e^{-\frac{b t}{m}})}{b} )</td>
<td>We enter an expression in the usual interactive two dimensional format. It's probably easier to enter the expression this way and see that it agrees with the mathematical conception.</td>
</tr>
<tr>
<td>( x(t) = \frac{m v0 x (1 + e^{-\frac{b t}{m}})}{b} )</td>
<td>We copy the expression and paste it into a separate place on the worksheet. We then select the expression and do Format-&gt;Convert To-&gt;1D Math Input. The result in red replaces the copy of the expression. We can then copy this into a code region.</td>
</tr>
<tr>
<td>( x0-m<em>v0</em>x*(-1+exp(-b*t/m))/b; )</td>
<td>We create a function definition involving this expression by entering the function name, the assignment operation :=, and the beginning of the function definition. Then we copy and paste in the 1D expression that we've manufactured.</td>
</tr>
</tbody>
</table>

9.13 Preventing catastrophic loss of worksheet information

The advice is: Save the current state of your work after making significant changes to the content of your worksheet.

When you start working for longer periods of time with computers, the unexpected may become more likely. Several of these unexpected scenarios could cause your work to be seriously disrupted.

1. There could be a power failure, or a system crash or freeze.

Saving your document before embarking on a test run will at least allow you to recover what you had before the crash. Crashes or freezes are not totally unheard in highly interactive programs with complicated interfaces, such as web browsers, Java-based applets, or systems such as Maple (whose interface is Java based). While you can try to recover from what Maple saved automatically while you were working (see the discussion in ????), saving a file yourself gives you explicit control over what you will recover after the crash.

2. Your most recent changes to your work are a big mistake. Suddenly you wish you could revert back to the way things were an hour ago, or yesterday.

Unless you are working in an environment that automatically preserves successive versions of your work, you will have to keep copies the old fashioned way -- by saving them under different names. One simple way is to append a version number, e.g. MyLab1-1.mw, MyLab1-2.mw, etc. If you aren't up to keeping track of the latest version number yourself, you could also use a time index label, e.g. MyLab1-103109-1014.mw.

3. Your computer could break down, making the files on it unavailable until the computer repaired, or even permanently unreadable should repairs not be possible. Computer theft is another way of losing the files on that computer permanently.
This doesn’t happen all that often, but probably happens to everyone who uses a computer for a few years. Hopefully you have saved copies of your valuable files elsewhere. This process is called creating a backup copy of your files, or just \textit{backing up}. In this era of “cloud computing” and inexpensive external storage (writable DVDs, external flash or hard drives), this is fairly easy to do for those who have the discipline to back up regularly. There are even some commercial programs (e.g. Mozy \url{https://mozy.com/registration}, SpiderOak \url{https://spideroak.com/}, or Macintosh Time Machine \url{http://www.apple.com/macosx/what-is-macosx/time-machine.html}, for example) that create “back up copies” automatically at periodic intervals without any effort on your part other than to make sure that the computer is powered on when the back ups should happen. The main point is to have the discipline to create backups before you end up in a situation where you have to work many weeks to recreate lost work.

Creating and having convenient access to back ups is expected for professionals who use computers. Losing the most recent version of a file may happen to everyone occasionally; losing a valuable file completely is as lame an excuse for a professional to use as “I forgot to do my homework” is for a student barring true calamities such as a natural disaster.

9.14 Summary of Chapter 9

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use of \textit{print}</strong></td>
<td></td>
</tr>
</tbody>
</table>
| \begin{verbatim}
L := \exp\left(-\frac{1}{10.0}\right), \exp\left(-\frac{2}{10.0}\right), \exp\left(-\frac{3}{10.0}\right) ;
\end{verbatim} | We set up a list of values. L has all floating-point values because the "10.0" in the denominator causes all computations to be done using limited-precision arithmetic. The display of the value of L is suppressed because we ended the assignment with a colon. |
| \begin{verbatim}
print("This list presents some values of the function \(e^x\), \exp(x));
print(". The last element of the list is ".L[-1]);
\end{verbatim} | print prints out the sequence of values given to it as arguments. You can give any number of items in the sequence. The commas separating the string and the number are from the sequence. Note that as first mentioned in XX, the result is not displayed because the line ends with a colon. |
| \begin{verbatim}
"This list presents some values of the function \(e^x\), \exp(x)".
The last element of the list is ".L[-1]";
\end{verbatim} | |
| \begin{verbatim}
print("There are ", \texttt{\textit{\textbf{\textcolor{red}{n}}}ops(L)},
" elements in this list. The largest value is ", max(L))
\end{verbatim} | |
| \begin{verbatim}
"There are ", \texttt{\textit{\textbf{\textcolor{red}{n}}} elements in this list. The largest value is ",max(L)\)
\end{verbatim} | (9.32) |

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use of \textit{printf}</strong></td>
<td></td>
</tr>
</tbody>
</table>
| \begin{verbatim}
L := \exp(-1.0), \exp(-2.0), \exp(-3.0), \frac{4}{5} ;
\end{verbatim} | We set up a list of values as in the previous example. |
| \begin{verbatim}
print("The last element of the list is \%d\,".L[-1])
\end{verbatim} | The first argument to \textit{printf} is a string describing how to format the rest of the values. The string contains format codes describing where to insert the values of the later arguments into the message described in the string. In this example, there is one format code \%f, for printing numbers in conventional decimal point notation. Since \textit{printf} joins together the message and the value together into a single message without a sequence into a single result, what is printed is without commas. |
| The last element of the list is 0.800000. | (9.34) |
| \begin{verbatim}
print("There are \%d\ elements in this list. \nThe largest is \%e\,". nops(L),
max(L))
\end{verbatim} | Additional \textit{printf} codes are \"\%d\" (for integer values), and \"\%e\" (for floating point numbers in scientific notation). The format code \texttt{\textit{\textbf{\textcolor{red}{n}}}in causes the next output to occur on the next line. |
| There are 4 elements in this list. The largest is 9.048374e-01. | (9.35) |
| \begin{verbatim}
print("L has \%d\ elements. \"max(L)\); printf("The largest is \%e\,".
max(L));
\end{verbatim} | Note that a \textit{printf} statement does not automatically cause the output on a new line. Thus, several |
printf statements executed together as these are will have all the output on a single line.

If you attempt to use %f format on an integer, or %d format on a number that is not an integer, you will get incorrect values (with no warning that they are incorrect), or possibly an error message. Let the user beware; the computer isn't going to necessary save you from making a mistake.

You can't print a list of floating point numbers using just %f format.

The %a format will handle arbitrary numerical or non-numeric values. The value being printed out is a list, which is a "non-numeric value" because it isn't a single number. However, the output uses the textual format rather than the "pretty" format used by print.

We set up a list of data points. Using sprintf, we create a string. The %d format is used to insert the number of points into the message.

We need the string to put a title onto our pointplot. You can read more about the title= argument to pointplot by entering plot, options in Maple's on-line help.

Insert -> Code Edit Region to create a region.
command-e (command-e in Macintosh) or
right-click -> Execute Region in a code edit region to execute all the code in it.
right-click→Component Properties to open a dialog box with the dimensions of the code region (in pixels). Change these dimensions by entering different numbers to enlarge or shrink the box. Sorry, dragging will not resize the code region.

right-click→Collapse Region to turn the code region into an icon.

Clicking on the icon will then execute the code inside.

The first line of code appears as the label for the icon.

right-click→Expand Region to turn a code region icon to display the region again.

### Semi-colons and colons

All Maple statements must be separated or terminated with either a semi-colon (;) or a colon (:). Usually each statement executed generates a line of output which appears in blue after the region. This sequence of output is called the execution trace of the code. If a colon is used, then the output that normally results from evaluating the statement is suppressed from the execution trace.

### Comments

Any portion of a line of code starting from a number sign (#) is treated as commentary and not as an operation to be performed.

### Example

```maple
# 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
```

Segments of lines that begin with # are regarded as program comments (for the program reader's eyes), not operations for Maple to carry out.

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 \textit{i})

Result of assignment to \textit{val}.

Result of assignment to \textit{term}.

Result of calling the \texttt{print} function.

Result of assignment to \textit{s}.

Result of assignment to \textit{tol}.

### Code Development Strategies

Create a code outline as a series of comments in a code region before you start coding. Refine the outline by adding subpoints until you have supplied enough details to write code.

Add \texttt{print} or \texttt{printf} statements to code in order to make the execution trace more intelligible and to make key results easier to find within it.

The goal in debugging is typically to inspect the execution trace and find the first place where things go wrong. Use this information to determine the line(s) of code you should look at to find and fix the causes of problems. In order to do this you need to know what is right, and you should create situations where you don’t have to look in too many places.

Develop incrementally – enter only a few lines of code at a time, and get them to work before adding more. Since comments always work, you can enter entire outlines without having to be particularly incremental.

Use an interactive worksheet to test out small segments of code before you enter them into a worksheet. This can be an easy way of entering a mathematical expression, for example.

Save your work to avoid losing it. Save your work using versioned names in order to avoid having new work mess up clean copies of things that used to work. Save your work on other computers to avoid losing it completely should your computer break.
10 Tables and other data structures

10.1 Chapter Overview

This brief chapter introduces Maple tables. Tables are another way in Maple to store multiple values in a single container. Like most other structures in Maple, they can hold any number of items.

10.2 A review of Maple data structures and properties

The value of computers in technical work comes not only from doing long calculations conveniently, but being able to do thousands or millions of shorter computations quickly. In order to harness this power without the programming becoming very tedious, we need to store inputs and outputs in data containers or data structures. We’ve already seen lists and sets as containers. Maple also has specialized containers used for particular purposes: ranges, equations, and expressions.

Let's review the data containers we've seen and used in Maple so far. Each kind of container has a different syntax to create it, and to access its parts. Variables can be assigned as their value their whole container. Some Maple functions use the whole container, while in other cases you use "parts access" to use particular items within the container. One common aspect of these data containers is that are immutable: once you've created them, you can't change them, you can only access (find out the value of) their value. In order to "change" a list or expression, for example, what you have to do is to create a new list/expression with the changes in it, and assign it to the same variable as had the old list.

<table>
<thead>
<tr>
<th>Type of container</th>
<th>Examples</th>
<th>How to create it</th>
<th>How to access a part of it</th>
<th>Operations that can be performed with container</th>
<th>Ways of changing data container after it's been created</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math expression</td>
<td>a+b+c exp(-b*x/3) sqrt(32)- y^2</td>
<td>Type it in using an arithmetic operation or mathematical function</td>
<td>op(i, expr) gets the i-th piece of the expression</td>
<td>Arithmetic and math functions: +, -, *, ^, /, sqrt, !, etc., nops(expr) tells you the number of terms in the expression.</td>
<td>Can't do it. Can only create a new container with edited content.</td>
</tr>
<tr>
<td>Range</td>
<td>1..10 as in plot(x^2, x=1..10)</td>
<td>Type it in using a &quot;..&quot;</td>
<td>op(1, r) gets the low end of the range, op(2,r) gets the high end</td>
<td>None other than access functions, but many functions use ranges as one of their parameters.</td>
<td>Can't do it.</td>
</tr>
</tbody>
</table>
| Equation          | 2*x-3=5 as in solve(2*x-3=5,x) | Type it in using a "=
| | | | rhs(eqn), lhs(eqn) gets the right and left hand sides of the equation | solve, fsolve, plot, etc. although usually other arguments are necessary | Can't do it. |
| List              | [1,1, 2.3, 5.5] | Type in items separated by commas, enclosed by square brackets. | L[i] gets the i-th item from the list. If i is negative, counting starts from the end. L[i..j] gets a list of the ith through jth items. | nops tells you the number of items in the list | Can't do it. |
| Sequences         | NULL | Type in items, separated by commas. | L[i] gets the i-th item from the list. If i is negative, counting starts from the end. L[i..j] gets a list of the ith through jth items. | All Maple functions accept sequences, although not all of them work with sequences that are the wrong length or provide the wrong kinds of values. | Can't do it. |
### 10.3 Tables

In this section, we introduce another kind of data structure, tables. A table is a structure where each element is accessed using a value called a *key* or *index*. Individual elements of the table can be assigned, in any order. Individual elements can be accessed using the index notation, e.g. \( t[2] \) to get the element of the table \( t \) whose access key is 2. This is similar to how lists worked, where \( L[2] \) for a list \( L \) was an expression whose value was the 2nd element of the list \( L \).

The *key* can be any valid Maple value, not necessarily an integer (although they often are). Tables can add elements over time with additional assignments.

Tables are useful in situations where we will be adding items to the data structure frequently, but we don't know in advance how many entries we will ultimately have. They will be more efficient to use than lists or sets in that kind of situation. In future chapters we will also see ways to exploit the other useful property of tables, where it is more convenient to have non-integer values for access keys.

#### Example

Demonstration of tables

```
restart

T := table( )  #empty set

Table:

(10.1) T := table( )

(10.2) T[1] := 3.5

(10.3) T[2] := 4.6


(10.6) T[ -1]

(10.7) indices(T)
```

- This creates a table named \( T \). Initially there are no items in the table. Whatever previous value \( T \) may have had is wiped out.
- Through assignment we create a table entry for \( T[1] \).
- Through assignment we create a table entry for \( T[2] \).
- Through assignment we create a table entry for \( T[4] \). The value of this entry depends on the values of \( T[1] \) and \( T[2] \).
- We reassign the value of \( T[2] \), overwriting the previous value.
- Expressions mentioning assigned elements of the table will use the assigned value. If that element of the table does not have an assigned value, it remains just as a symbol.
- The *indices* function will return a sequence of the access keys for the table. Each access key is enclosed in brackets so that you can tell where one key stops and the next begins.
### Example

| entries(T) | \[3.5, 4.7, 17.3\] | The entries function will return a sequence of values stored in the table. The sequence is generally listed in the same order that the indices function lists access keys. (10.8) |
| \n | \n | \n |
| nops([indices(T)]) | 4 | This is the way to tell how many items are in a table at any time. (10.9) |
| L := convert(T, list) | \[3.5, 4.7, 17.3\] | The convert command can be used to put all the entries of a table into one list. (10.10) |
| T[x] := 92·exp(-3·t) | \(92\cdot e^{-3t}\) | The indices of a table can also be letters. Here we assign a value to the index \(T[x]\). (10.11) |
| indices(T) | \[1, 2, 4, x\] | (10.12) |
| entries(T) | \[3.5, 4.7, 17.3, 92\cdot e^{-3t}\] | (10.13) |
| T[2] := 'T[2]' | \(T_2\) | This is the way to remove an entry from the table -- putting quotes (', not backquotes `) around the item being removed. This is also the way unassignment of values were done for ordinary variables in section 4.2. (10.14) |
| indices(T) | \[1, 4, x\] | (10.15) |
| Lnew := convert(T, list) | \[3.5, 17.3, 92\cdot e^{-3t}\] | (10.16) |
| S := table([1 = 2, 2 = 4, 3 = 9]) | table([1 = 2, 2 = 4, 3 = 9]) | If we want to fill up a table by giving a multiple entries at once, we can give the table function a list of equations indicating indices and the associated value. (10.17) |
| S[1] | 2 | (10.18) |

There are three major differences between tables in Maple and the data containers we've seen previously: a) there is a separate creation step (creating a table), b) you can change the contents of the table be assigning, c) items within a table are stored under a key or index value. The latter works similar to the indexing used for lists, but the index does not have to be an integer. A table can store any kind of item under any kind of key. For the time being, we will stick with integer keys, but eventually we will take greater advantage of this "key" feature.
## 10.4 Troubleshooting with tables

1. Table creation `tbl := table( );` should always appear at the beginning of a script, which assigns `tbl` a fresh, empty table. If you don't do this, you may see items left over in `tbl` from the last time you executed a script in the same worksheet with `tbl` in it. Alternatively, you can do a `restart;` before running a script that uses a table.

2. Use entries and indices to see what's in your table. Check the results to make sure that there are no unexpected indices and that the values are computed correctly.

3. Know when to use square braces and when to use parentheses in working with tables.

<table>
<thead>
<tr>
<th>Troubleshooting with tables: mistaking [ ] and ()</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>T := table( );</code></td>
</tr>
<tr>
<td><code>T(3) := 4.7</code></td>
</tr>
<tr>
<td>Error, invalid left hand side in assignment</td>
</tr>
<tr>
<td><code>T[3] := 4.7</code></td>
</tr>
<tr>
<td><code>T[3]</code></td>
</tr>
<tr>
<td><code>T(3)</code></td>
</tr>
<tr>
<td><code>T := table( )</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Troubleshooting tables: checking for extra keys</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>T := table( )</code></td>
</tr>
<tr>
<td><code>T[1] := 4.7</code></td>
</tr>
<tr>
<td><code>T[2] := 4.8</code></td>
</tr>
<tr>
<td><code>T[3] := 4.9</code></td>
</tr>
<tr>
<td><code>T[3]</code></td>
</tr>
</tbody>
</table>

### Troubleshooting Guidelines

- **Table Creation:** Always use `table( );` at the beginning of a script to ensure a fresh, empty table.
- **Entry Placement:** Use `table[ ]` to access table values. Avoid using parentheses (`( )`).
- **Checking for Keys:** Be aware of potential confusion when keys can be any value. Use table keys appropriately.
- **Assignments:** Ensure that you are assigning to the correct index in your table.

---

142 • 10 Tables and other data structures
### Troubleshooting tables: checking for extra keys

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f := 6 - i )</td>
<td>( 4 )</td>
</tr>
<tr>
<td>( T[k] := 5.0 )</td>
<td>( 5.0 )</td>
</tr>
<tr>
<td>( T[j] := 5.1 )</td>
<td>( 5.1 )</td>
</tr>
<tr>
<td>( T[1] := 4.75 )</td>
<td>( 4.75 )</td>
</tr>
<tr>
<td>( T[1], T[2], T[3], T[4] )</td>
<td>( 4.75, 4.8, 4.9, 5.1 )</td>
</tr>
</tbody>
</table>

Where did the 5.0 go? We thought we did \( T[3.0] := 5.0 \) with the use of \( k \)

**indices(\( T \))**

\[ [1], [2], [3], [4], [3.0] \]

**entries(\( T \))**

\[ [4.75], [4.8], [4.9], [5.1], [5.0] \]

We use the \( \text{indices} \) and \( \text{entries} \) functions to tell us what is going on in \( T \). Evidently the keys used for \( T \) include both 3 and 3.0. Dimly, light comes on where we recall that 3 and 3.0 are regarded as different in Maple. "3" is an exact integer and "3.0" is a floating point number.

\( T[3.0] \)

\( 5.0 \)

\( T[3] \)

\( 4.9 \)

\( T[3] := 5.0 \)

\( 5.0 \)

**indices(\( T \))**

\[ [1], [2], [3], [4], [3.0] \]

**entries(\( T \))**

\[ [4.75], [4.8], [5.0], [5.1], [5.0] \]

We see that there are two distinct entries in the table. Reassigning one of them does not erase the other entry, although the value of the first one does change.

\( T[3.0] := T[3.0] \)

\( T_{3.0} \)

**indices(\( T \))**

\[ [1], [2], [3], [4] \]

As indicated in the demonstration of tables (Brian: provide hyperlink to earlier in the chapter), unassigning a table entry will erase it from the table.
### 10.5 Summary of Chapter 10

<table>
<thead>
<tr>
<th>Type of container</th>
<th>Examples</th>
<th>How to create it</th>
<th>How to access a part of it</th>
<th>Operations that can be performed with container</th>
<th>Ways of changing data container after it’s been created</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>T := table(); T[1] := 32.5; T[2] := 32.6;</td>
<td>tblName := table(); This creates an empty table and assigns it to the variable tblName replacing whatever previous value it had.</td>
<td>T[key] retrieves the value stored under the key from within the table. Thus if you have done T[1] := 32.5; then subsequently doing T[1] will retrieve the 32.5. The key can be any value Maple expression. Thus x, or t=3..5 can be keys for a Table.</td>
<td>entries(T) provides a sequence of the values stored in the table (including duplicate values) indices(T) provides a sequence of the keys used by the table.</td>
<td>Key-value assignment: T[1] := 32.7</td>
</tr>
</tbody>
</table>
11 For Looping: repetitive execution

11.1 Chapter Overview

The computer can carry out thousands or millions of instructions in Maple per second. While a "powerful" instruction such as `solve` or `plot` to Maple is a way of having a few lines of code tap into this power, we need a way of specifying calculations that perform a lot of work without writing a commensurate amount of code. Maple's `for...do...end do` and `while...do...end do` instructions are a way of specifying that lines of code be repeated a definite number of times, or indefinitely until some condition is no longer true. This chapter the syntax -- how to say it -- and some example calculations that use this capability.

Such programming is called looping because the computer is told to loop back and repeat some instructions. Loop programming is a basic skill all programmers need to know, since most programming languages have similar ways of doing repetition.

11.2 Repetition with to ... do .... end do

In programming it's common to want to repeat a segment of a code region a few times. We could do that by copying and pasting the code that many times, but this gets tedious. We might also want to place the repetition factor under control of a script parameter, so that it would be easy to change the number of times things could be repeated.

Maple provides a convenient way to control the repetition of a code segment through something called a loop. The segment of code that is repeated is called the loop body. Its general form looks like this:

to \( n \) do
  segment of code
end do;

where \( n \) is an expression describing how many times the segment should be repeated.

Examples of repetitive looping with to...do... end do

<table>
<thead>
<tr>
<th>Examples of repetitive looping with to...do... end do</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>to 5 do</td>
<td></td>
</tr>
<tr>
<td>printf(&quot;I will not burp in class.\n&quot;);</td>
<td></td>
</tr>
<tr>
<td>end do;</td>
<td></td>
</tr>
<tr>
<td>printf(&quot;The repetition is over.\n&quot;):</td>
<td></td>
</tr>
<tr>
<td>I will not not burp in class.</td>
<td></td>
</tr>
<tr>
<td>I will not not burp in class.</td>
<td></td>
</tr>
<tr>
<td>I will not not burp in class.</td>
<td></td>
</tr>
<tr>
<td>I will not not burp in class.</td>
<td></td>
</tr>
<tr>
<td>I will not not burp in class.</td>
<td></td>
</tr>
<tr>
<td>The repetition is over.</td>
<td></td>
</tr>
</tbody>
</table>

\( n := 7; \)

to \( n \) do
  printf("I will write %d times, "Underwear should be worn on the inside.\"\n", n);
end do;
printf("The repetition is over.\n"): 7

The 'n at the end of the printf is necessary or else all the output will appear on the same line.

Recall that the execution trace contains both the results of each line execution, and the output of any print or printf statements that are being executed. Since the ??? which displays as "nothing", this execution trace contains just the side effect of displaying the various messages.

For these and other repetitive sayings by Bart Simpson, see http://www.snpp.com/guides/chalkboard.openings.html.

This script makes it easy to change the number of times the loop repeats itself, through the standard act of changing the value of a parameter at the beginning of the script.
I will write 7 times, "Underwear should be worn on the inside."
I will write 7 times, "Underwear should be worn on the inside."
I will write 7 times, "Underwear should be worn on the inside."
I will write 7 times, "Underwear should be worn on the inside."
I will write 7 times, "Underwear should be worn on the inside."
I will write 7 times, "Underwear should be worn on the inside."
The repetition is over.

| n := 7; 
| to n do 
| printf("I will write %d times, "Underwear should be worn on the inside."\"\n", n); 
| printf("The repetition is over.\n"); |

Troubleshooting loops: if the "end do" is left out -- you may see a warning and more importantly, none of the expected repetitions. Furthermore, none of the post-repetition actions seem to have been performed.

It's stylistic convention to indent the loop body code although not required by the grammatical rules of Maple. This is because it makes it easier to determine the extent of loops particularly in more complicated codes where there are multiple loops or even nested loops -- loops within loops.

### 11.3 Repetition with from ... to ... do ... end do and from ... to ... by .... do ... end do

We can introduce a "stepper variable" into the repetition through the use for stepperVariable from initial value to final value do .... end do;

Sometimes the stepper variable is referred to as an index variable.

The first time the code segment is executed, the value of the stepper variable is initial value. The second time, it is set to one more, initial value +1. etc. The last repetition is typically final value.

Typically, initial value and final value are expressions that evaluate to integers (whole numbers).

#### Example of repetition with a stepper/index variable: for ... from .. to ... do ... end do

Maple, having Algol 68 in its ancestry, allows loop bodies to end with "od" (do spelled backwards) as well as the more English like "end do".
evalf(sin(i)))
od;
printf("The repetition is over.\n"):

Repetition 1:  sin(1) = 0.841471.
Repetition 2:  sin(2) = 0.909297.
Repetition 3:  sin(3) = 0.141120.
Repetition 4:  sin(4) = -0.756802.
Repetition 5:  sin(5) = -0.958924.
The repetition is over.

We can make the stepperVariable take on different values between initialValue and finalValue through the use of a stepValue.

for stepperVariable from initialValue to finalValue by stepValue do ... end do
will repeat the loop.

Example of repetition with a stepper/index variable: for ... from .. to ... by ... do ... end do

Example of repetition with a stepper/index variable: for ... from .. to ... by ... do ... end do

This kind of loop can be used to print out tables of values, where each row depends on the value of the stepper variable.

What would we have to change in order to print values of 
sin for 0.0, 0.05, 0.10, 0.15?

for x from 0 by .1 to fv do
  printf("sin(%f) = %f.\n", x, evalf(sin(x)))
end do;
printf("The repetition is over.\n"):  

sin(0.000000) = 0.000000.
sin(0.100000) = 0.099833.
sin(0.200000) = 0.198669.
sin(0.300000) = 0.295520.
sin(0.400000) = 0.389418.
sin(0.500000) = 0.479426.
sin(0.600000) = 0.564642.
sin(0.700000) = 0.644218.
sin(0.800000) = 0.717356.
sin(0.900000) = 0.783327.
sin(1.000000) = 0.841471.
sin(1.100000) = 0.891207.
sin(1.200000) = 0.932039.
The repetition is over.

This kind of loop can be used to print out tables of values, where each row depends on the value of the stepper variable.

What would we have to change in order to print out values of since for 0.0, 0.05, 0.10, 0.15?

for x from fv to 0 by -.1 do
  printf("exp(%f) = %f.\n", x, evalf(exp(x)))
end do;

exp(1.100000) = 3.001424.
exp(1.000000) = 2.718282.
exp(0.900000) = 2.344213.
exp(0.800000) = 1.976208.
exp(0.700000) = 1.648721.
exp(0.600000) = 1.349857.
exp(0.500000) = 1.071699.
exp(0.400000) = 0.825395.
exp(0.300000) = 0.606531.
exp(0.200000) = 0.411516.
exp(0.100000) = 0.264879.
exp(0.000000) = 0.000000.
The repetition is over.

We can make the stepper value decrease in values with each repetition by having the starting value (in this case, 1.2) larger than the final value (in this case, 0), and making the step value negative (-.1).
11.4 Repetition with for... from ... to ... by .... do ... end do

General form:

for stepperVariable from initialValue to finalValue by stepValue do loop body;
end do;

We can make the stepperVariable take on different values between initialValue and finalValue through the use of a stepValue. We can even make the stepperVariable decrease rather than increase by giving a negative value for the stepValue.

For loop with a different step size

```
for x from 0 by .1 to 1 do
    printf("\sin(%f) = %f.\n", x, evalf(sin(x)));
end do;
```

<table>
<thead>
<tr>
<th>x</th>
<th>\sin(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>0.100000</td>
<td>0.099833</td>
</tr>
<tr>
<td>0.200000</td>
<td>0.198669</td>
</tr>
<tr>
<td>0.300000</td>
<td>0.295520</td>
</tr>
<tr>
<td>0.400000</td>
<td>0.389418</td>
</tr>
<tr>
<td>0.500000</td>
<td>0.479426</td>
</tr>
<tr>
<td>0.600000</td>
<td>0.564642</td>
</tr>
<tr>
<td>0.700000</td>
<td>0.644218</td>
</tr>
<tr>
<td>0.800000</td>
<td>0.717356</td>
</tr>
<tr>
<td>0.900000</td>
<td>0.783327</td>
</tr>
<tr>
<td>1.000000</td>
<td>0.841471</td>
</tr>
</tbody>
</table>

The from, to, and by portions of a for are all optional. If that portion is skipped, then 1 is used for the initialValue, finalValue, or stepValue. We can give the information in any order and it will mean the same thing. Thus "for i from 1 to 10 by 3..." is as acceptable as "for i to 10 by 3 from 1 do".

For loop with a decrementing rather than incrementing steps

```
for x by -.3 to -1 do
    printf("\sqrt(%f) = %a\n", x, evalf(sqrt(x)));
end do;
```

<table>
<thead>
<tr>
<th>x</th>
<th>\sqrt(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.000000</td>
<td>1.000000</td>
</tr>
</tbody>
</table>
sqrt(0.700000) = .8366600265
sqrt(0.400000) = .6324555320
sqrt(0.100000) = .3162277660
sqrt(-0.200000) = .4472135955*I
sqrt(-0.500000) = .7071067812*I
sqrt(-0.800000) = .8944271910*I

We are using '%a format rather than '%e or '%f format because some of the answers are imaginary numbers, which are not floating point so don't work with the other formats.

Since there is no "from" in the for statement, the initial value of x is 1. Since the step size is -0.3, then the final value should be less than initial value of 1.

### 11.5 A pattern of looping: accumulation of results

So far we have seen examples of loops where what happens inside the loop is based on the current value of the index. However, by doing a little extra work before and during the loop, we can calculate accumulative results.

For example, suppose we want to add together the first 100 numbers.

We could start typing in the single expression:

\[
result := 1 + 2 + 3 + 4 + 5 + \ldots \]

but we can see that this is going to be a lot of work. We can do this with a loop by using the ability of variables to add onto their present value. What is required to initialize the variable, and then repetitively accumulate.

**An example of an accumulation loop**

```plaintext
#Set parameters of script
n := 10;

#initialize accumulation variable
result := 0;

#repeat: add i to the accumulated sum
for i to n do
    result := result+i;
end do;

printf("The sum of the numbers from 1 to \%d is: \%d", n, result);
```

Output:

```
10
0
1
3
6
10
15
21
```
The sum of the numbers from 1 to 10 is: 55

We see that the execution trace shows \( n \) and \( \text{result} \) being assigned its pre-loop initial value. With each repetition of the loop, result is changed, with the current value of the stepper variable \( i \) added onto it. In the first repetition, \( i \)'s initial value is added onto result. In the next repetition, \( i \) has a value of 2, so 2 is added onto result (producing 3). This continues until the last repetition, where \( i \)'s final value of 10 is added onto result.

We don't have to accumulate results in sums. For example, we can accumulate results in a table, or multiple tables.

Accumulating results in two tables and plotting them

```plaintext
# Set parameters of script
n := 5;

# Initialize accumulation variables (empty tables)
yVals := table();
tVals := table();

# Repeat: calculate \( 10^i \) and its algorithm. Store in table
# do this for \( i = 1, 2, 3, 4, 5 \).
for i to n do 
    # There is no "from" given. It's optional.
    tVals[i] := 10^i;
    yVals[i] := evalf(log(tVals[i]));
    printf("Logarithm of %.10f is %.5f\n", tVals[i], yVals[i]);
end do;

# After the loop is finished, convert tables into a list
tList := convert(tVals,list);
yList := convert(yVals,list);

# Do a data plot
plot(tList,yList, style=point, symbol=diamond, color=red, labels=["x","log(x)"]);
```

Logarithm of 10.000000 is 2.302585
11.6 Suppressing execution trace output inside loops

Note that we can make the script add the numbers from 1 to 1000 by changing the one line defining the value of \( n \). However, the output grows somewhat tedious to view. We can suppress the part of the execution trace by putting a colon at the end of the "end do".

Accumulation of two results in the same loop; suppression of output

<table>
<thead>
<tr>
<th>Accumulation of two results in the same loop; suppression of output</th>
<th>What the execution trace of this loop looks like without output suppression. Since the loop computes two things per repetition, and repeats 10 times, there are 10 lines of output to look at. Imagine what would happen if there ( n ) were 100.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n := 5; )</td>
<td>( n := 5; )</td>
</tr>
<tr>
<td>result := 0;</td>
<td>result := 0;</td>
</tr>
<tr>
<td>result2 := 1;</td>
<td>result2 := 1;</td>
</tr>
<tr>
<td>for i to n do</td>
<td>for i to n do</td>
</tr>
<tr>
<td>Logarithm of 100.000000 is 4.605170</td>
<td>Logarithm of 100.000000 is 4.605170</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>4.605170186</td>
<td>4.605170186</td>
</tr>
<tr>
<td>Logarithm of 1000.000000 is 6.907755</td>
<td>Logarithm of 1000.000000 is 6.907755</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>6.90775279</td>
<td>6.90775279</td>
</tr>
<tr>
<td>Logarithm of 10000.000000 is 9.210340</td>
<td>Logarithm of 10000.000000 is 9.210340</td>
</tr>
<tr>
<td>10000</td>
<td>10000</td>
</tr>
<tr>
<td>9.210340372</td>
<td>9.210340372</td>
</tr>
<tr>
<td>Logarithm of 100000.000000 is 11.512925</td>
<td>Logarithm of 100000.000000 is 11.512925</td>
</tr>
<tr>
<td>100000</td>
<td>100000</td>
</tr>
<tr>
<td>11.51292546</td>
<td>11.51292546</td>
</tr>
<tr>
<td>[10, 100, 1000, 10000, 100000]</td>
<td>[2.302585093, 4.605170186, 6.90775279, 9.210340372, 11.51292546]</td>
</tr>
</tbody>
</table>
result := result+i;
result2 := result2*i^2;
end do;
printf("The sum of the numbers from 1 to %d is: %d\n", n, result);
printf("The product of squares i^2 from i=1 to %d is %d\n", n, result2);

5
0
1
1
1
3
4
6
36
10
576
15
14400

The sum of the numbers from 1 to 5 is: 15.
The product of squares i^2 from i=1 to 5 is 14400.

n := 10;
result := 0;
result2 := 1;
for i to n do
  result := result+i;
  result2 := result2*i^2;
end do:
printf("The sum of the numbers from 1 to %d is: %d\n", n, result);
printf("The product of squares i^2 from i=1 to %d is %d\n", n, result2);

10
0
1

We suppressed output from the execution trace by placing a colon at the end do, which suppresses the execution trace of everything in the loop body. This is something to do after you have gotten the loop to work correctly.
The sum of the numbers from 1 to 10 is: 55.
The product of squares $i^2$ from $i=1$ to 10 is $13168189440000$.

11.7 Application of accumulative looping: making plots and movies with lists and tables

Plot the position of a particle moving under Henon’s rules of motion

```plaintext
# Henon attractor
# Example of a "chaotic dynamical system"

# Governed by the equations of motion where
# next value of $x$ = present value of $y + 1 - 1.3 \times (\text{present value of } x)^2$
# next value of $y = 0.3 \times (\text{present value of } x)$

# We start with $x$ and $y$ initially 0.
# We store successive values in a table for $x$ and $y$ values.
# The keys (indices) are the steps $i = 0, 1, 2, 3, ...$

# Initialize $x$ and $y$ tables
x := table():
y := table():
x[0] := 0;
y[0] := 0;

# Take 500 time steps through the system
for i from 0 to 4999 do
    x[i+1] := y[i] + 1 - 1.3 \times (x[i])^2;
y[i+1] := 0.3 \times x[i];
end do:  # Suppress output of loop (because we've already tested it)

xL := convert(x,list):
yL := convert(y,list):
printf("Size of list :%d\n",nops(xL));
plot(xL,yL, color=red);

Size of list :5001
```
The accumulation variables are the two tables x and y. As the loop repeats the tables fill with successive computed values. Even though the motion is circumscribed, the motion of the particle cannot be described by a simple mathematical formula such as the models we used for trajectories of a Human cannonball.

Example: make a movie of the position of a particle moving under Henon's rules of motion

```plaintext
Henon attractor
Example of a "chaotic dynamical system"

# Governed by the equations of motion where
# next value of x = present value of y + 1 - 1.3*(present value of x)^2
# next value of y = 0.3*(present value of x)

# We start with x and y initially 0.
# We store successive values in a table for x and y values.
# The keys (indices) are the steps i = 0, 1, 2, 3....

# Initialize x and y tables
x := table();
y := table();
p := table();
x[0] := 0;
y[0] := 0;
p[0] := plot([x[0]], [y[0]], style=point, color=red);

# Take 500 time steps through the system
for i from 0 to 49 do
    x[i+1] := y[i] + 1 - 1.3*(x[i])^2;
y[i+1] := .3*x[i];
p[i+1] := plot([x[i+1]], [y[i+1]], style=point, color=red);
end do:  # Suppress output of loop (because we've already tested it)

pl := convert(p,list):
printf("Size of list :%d\n", nops(pl));
```
with(plots):
display(pl,insequence=true);

```
table([])
  0
  0
```

Size of list :51

The accumulation variables are the two tables x and y as in the previous example, along with the table p that contains a point plot for each value of the index i. At the end, instead of printing or plotting the values of x and y, the insequence option of display is used to take the list of point plots and turn them into an animation. Compared to the graph, the animation is another way of gaining insight into how the particle moves over the time.

11.8 Troubleshooting with loops

Some of the basic problems come from forgetting the "end do" or putting too many in. Examples of this were given (page 146).

Another problem with a loop is that the values provided for its control cause a different effect than you expected. Discover the difference between what the rules say will happen and what you thought would happen by observing the execution trace, possibly putting in print statements.

Finally, in accumulation loops, the initial value of the accumulator variable may be wrong, or the accumulation is not happening as planned. The control of the loop may stop the accumulation too early, or too late.

Troubleshooting loops: a script that doesn't work

```
s := 0;
x := .2;
#Approximate exp(x) = 1 + x + x^2/2! + x^3/3! + ...
for i to 10 from 1 by 1 do
  #We changed the order of to and from, but that's not an error
  s := s + x^i/i!;
```
end do;
printf("%f compared to expected %f", s, exp(x));

0
0.2
0.2
0.220000000
0.221333333
0.221400000
0.221402667
0.221402756
0.221402758
0.221402758
0.221402758
0.221402758

0.221403 compared to expected 1.221403

The execution trace seems to indicate that the right number of terms is being computed, but the answer we computed by our own programming is different than what Maple's built-in \texttt{exp} function produces. The computed answer seems to be one less than the answer we expect. This is different than the other examples previously shown where the programmed results agreed quite closely with Maple's built-in computation.

\textbf{Rerunning the script with more information generated by the execution trace}

\begin{verbatim}
s := 0;
x := .2;
#Approximate \texttt{exp}(x) = 1 + x + x^2/2! + x^3/3! + ...

for i to 10 from 1 by 1 do
  printf("i= %d, x = %f, s = %f, x^i/i! = %f\n", i, x, s, x^i/i!);
  s := s + x^i/i!;
end do;
printf("%f compared to expected %f", s, exp(x));
\end{verbatim}

We enhance the execution trace by having it print out the values of all the variables with each repetition. It's work to enter in this printf statement, but it pays off in helping us find the mistake more quickly. Comparing the actual value of \texttt{s} with what the program comment tells us to expect term by term, we see that while \texttt{s} starts off at zero in the first iteration, the first term \(\frac{x}{0!}\) is .2, not 1 as we expect. Looking at the second and third terms, we realize that we have forgotten to add in the first term to \texttt{s}.

There are two ways to do this -- either to start at \texttt{i=0} (since \(\frac{x}{0!}\) is 1) or to keep the loop as it is and just initialize \texttt{s} to 1 instead of zero before the loop starts. We opt for the latter since it doesn't require remembering facts from high school days such as that \(0!=1\).

\textbf{Rerunning the script with repairs}
\[
s := 1;
x := .2;
\]

& Approximate \(\exp(x) = 1 + x + x^2/2! + x^3/3! + \ldots\)

\[
\text{for } i \text{ to 10 from 1 by 1 do}
\]
\[
\begin{align*}
\text{printf(} & "\ i = \ %d, \ x = \ %f, \ s = \ %f, \ x^i/i! = \ %f\n", \ i, \ x, \ s, \ x^i/i!); \\
\text{s := s + x^i/i!;}
\end{align*}
\]
\[
\text{end do;}
\]
\[
\text{printf(} "\ %f \text{ compared to expected } \ %f", \ s, \ \exp(x));
\]

<table>
<thead>
<tr>
<th>i</th>
<th>(x)</th>
<th>(s)</th>
<th>(x^i/i!)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.2</td>
<td>1.0</td>
<td>0.200000</td>
</tr>
<tr>
<td>2</td>
<td>0.2</td>
<td>1.20</td>
<td>0.020000</td>
</tr>
<tr>
<td>3</td>
<td>0.2</td>
<td>1.22</td>
<td>0.001333</td>
</tr>
<tr>
<td>4</td>
<td>0.2</td>
<td>1.221</td>
<td>0.000067</td>
</tr>
<tr>
<td>5</td>
<td>0.2</td>
<td>1.2214</td>
<td>0.000003</td>
</tr>
<tr>
<td>6</td>
<td>0.2</td>
<td>1.2214</td>
<td>0.000000</td>
</tr>
<tr>
<td>7</td>
<td>0.2</td>
<td>1.2214</td>
<td>0.000000</td>
</tr>
<tr>
<td>8</td>
<td>0.2</td>
<td>1.2214</td>
<td>0.000000</td>
</tr>
<tr>
<td>9</td>
<td>0.2</td>
<td>1.2214</td>
<td>0.000000</td>
</tr>
<tr>
<td>10</td>
<td>0.2</td>
<td>1.2214</td>
<td>0.000000</td>
</tr>
</tbody>
</table>

1.221403 compared to expected 1.221403

Making this change produces the expected agreement. At this point, we would "comment out" the printf that we put in to enhance the execution trace, and could change the value of \(x\) to compute an approximation to \(\exp(x)\) for other values of \(x\).

More pernicious are loops which do not give error messages but either stop with the wrong answer, or do not stop.

Runaway loops can be stopped by hitting the red "stop hand". But the result is not really satisfactory, as illustrated in the example below.
A runaway "infinite" loop

```plaintext
for x from 3.0 by .1 do
    print("sin(x)", sin(x));
end do;
```

"sin(x)", 3.0, 0.1411200081 "sin(x)", 3.1, 0.04158066243 "sin(x)", 3.2, -0.05874143434 "sin(x)", 3.3, -0.1577456941 "sin(x)", 3.4, -0.2555411020 "sin(x)", 3.5, -0.3507832277 "sin(x)", 3.6, -0.4425204433 "sin(x)", 3.7, -0.5298361409

e tc., etc.

The problem with this loop is that it will never stop. Because we mistakenly left off the "to finalValue" portion of the loop (which is legal, even if dangerous), the loop will just keep repeating, adding .1 onto the value of x. Because the loop body never changes the value of x, it will continue to be true. Presumably the programmer made a mistake and forgot to put in lines of code into the loop body that changed the value of x and (with enough repetition) made it eventually grow to 10 or larger.

While in principle the "red stop hand" is useful for computations that don't do a lot of printing, in practice it is bad news if you make a mistake and execute this kind of infinite loop. Like the Sorcerer's Apprentice (see http://en.wikipedia.org/wiki/The_Sorcerer%27s_Apprentice if you don't know the cultural reference), you will see seemingly endless lines of output. Clicking on the red stop hand will eventually stop the output, but not until after a long, long wait while all the output that resulted from before you used the hand scrolls by. This is because Maple typically generates thousands of repetitions of the loop in less than a second. It takes a long time for it to print all the output from the repetitions, and you won't be able shut off the printing with the red hand.

About the only way to make the output stop is to force quit out of the Maple application. This will cause you to lose the current state of the worksheet!

If you've saved your worksheet while you were working, then you can recover something close to the state before the "infinite loop" made a mockery of your session. Alternatively, you can recover the worksheet using File->Recent Documents -> Restore ??? as described in ???.

### 11.9 Summary of Chapter 11

<table>
<thead>
<tr>
<th>Repetition type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repetition with to ... do ... end do</td>
<td></td>
</tr>
<tr>
<td></td>
<td>to 5 do</td>
</tr>
<tr>
<td></td>
<td>printf(&quot;I will not not burp in class.\n&quot;);</td>
</tr>
<tr>
<td></td>
<td>end do;</td>
</tr>
<tr>
<td></td>
<td>printf(&quot;The repetition is over.\n&quot;);</td>
</tr>
<tr>
<td>Repetition with from ... to ... do ... end do</td>
<td>#Set parameters</td>
</tr>
<tr>
<td></td>
<td>n := 5;</td>
</tr>
<tr>
<td></td>
<td>#Repeat an action with i stepping through the values</td>
</tr>
<tr>
<td></td>
<td>#1, 2, 3, 4, ... n</td>
</tr>
<tr>
<td></td>
<td>for i to n do</td>
</tr>
<tr>
<td></td>
<td>printf(&quot;Repetition %d:  sin(%d) = %f.\n&quot;,</td>
</tr>
<tr>
<td></td>
<td>i, i, evalf(sin(i)))</td>
</tr>
<tr>
<td></td>
<td>od;</td>
</tr>
<tr>
<td></td>
<td>printf(&quot;The repetition is over.\n&quot;);</td>
</tr>
<tr>
<td>Repetition with from ... to ... by ... do ... end do</td>
<td>#Final value parameter</td>
</tr>
<tr>
<td></td>
<td>fv := 1.2;</td>
</tr>
<tr>
<td></td>
<td>#Repeat an action with i stepping through the values</td>
</tr>
<tr>
<td></td>
<td>#0, .1, .2, .3, ....</td>
</tr>
<tr>
<td></td>
<td>for x from 0 by .1 to fv do</td>
</tr>
</tbody>
</table>
Repetition (incrementing) with \texttt{for ... from ... to ... by ... do ... end do}:
\begin{verbatim}
for x from 0 by .1 to 1 do
  printf("sin(%f) = %f.\n", x, evalf(sin(x)))
end do;
\end{verbatim}

Repetition (decrementing) with \texttt{for ... to ... do ... end do}:
\begin{verbatim}
for x by -.3 to -1 do
  printf("sqrt(%f) = %a\n", x, evalf(sqrt(x)))
end do;
\end{verbatim}

Accumulation loop type  

<table>
<thead>
<tr>
<th>Prints result</th>
<th>Example</th>
</tr>
</thead>
</table>
| #Set parameters of script  
n := 10;  
#initialize accumulation variable  
result := 0;  
#repeat: add i to the accumulated sum  
for i to n do  
  result := result+i;  
end do;  
printf("The sum of the numbers from 1 to 1\n%d is: %d", n, result); |

Storing results in a table  

<table>
<thead>
<tr>
<th>Storing results in a table</th>
<th>Example</th>
</tr>
</thead>
</table>
| #Set parameters of script  
n := 5;  
#initialize accumulation variables (empty tables)  
yVals := table();  
tVals := table();  
#repeat: calculate 10^i and its algorithm. Store in table  
#do this for i = 1, 2, 3, 4, 5.  
for i to n do  
  #there is no "from" given.  
  #there is no "from" given.  
  tVals[i] := 10^i;  
  yVals[i] := evalf(log(tVals[i]));  
  printf("Logarithm of %f is %f\n",tVals[i], yVals[i]);  
end do;  
#After the loop is finished, convert tables into a list |
tList := convert(tVals, list);
yList := convert(yVals, list);

# Do a data plot
plot(tList, yList, style=point, symbol=diamond, color=red, labels=['x', 'log(x)']);

Accumulating two results in the same loop, suppressing output
n := 5;
result := 0;
result2 := 1;
for i to n do
    result := result + i;
    result2 := result2 * i^2;
end do;
printf("The sum of the numbers from 1 to %d is: %d.
", n, result);
printf("The product of squares i^2 from i=1 to %d is %d.
", n, result2);

Troubleshooting tips
Example
Print out values of variables using execution trace
s := 1;
x := .2;
#Approximate exp(x) = 1 + x + x^2/2! + x^3/3! + ...
for i to 10 from 1 by 1 do
    printf("i= %d, x = %f, s = %f, x^i/i! = %f
", i, x, s, x^i/i!);
    s := s + x^i/i!;
end do;
printf("%f compared to expected %f", s, exp(x));

Do not create an infinite loop
# Do not run this
# We do not have a "to finalValue" portion in our for # loop
for x from 3.0 by .1 do
    printf("sin(x)", sin(x));
end do;
12 More on repetition and looping

12.1 Chapter Overview

In the previous chapter we have seen how for...do...end do can be used to repeat a segment of code, possibly with a stepping variable that is automatically managed by Maple during the execution. In this chapter we look at other ways of getting repetition to occur: through some of the built-in shortcuts such as seq, sum, and addition of lists.

We also look at while .. do...end do which is a way of having repetition occur while a condition holds. for ... while ... do .... end do is a way of having both a stepping variable with an upper limit, and an additional condition that can end the repetition early if it becomes false.

12.2 Implicit looping: seq, list addition, sum

In chapter 11, we learned about for and its ability to loop with an index (stepping) variable. While this is a marvelous abbreviation, it still takes some more work to set up. Certain repetitious actions occur so commonly that many programming languages provided even shorter ways of performing looping idioms.

For example, suppose we wanted to build a list consisting of the numbers 1, 4, 9, 25, ... 169. We could type such a list in, but from what we know of looping we suspect that there’s an easier way. Using the ideas we’ve seen so far, we could use a table valueTab that has an accumulation variable L. With i as the stepping variable going from 1 to 100, we would fill up the table with $i^2$ values. After the loop finishes, we could convert the table into a list to get our result:

Making a list through a loop

<table>
<thead>
<tr>
<th>Making a list through a loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>#initialize table</td>
</tr>
<tr>
<td>valueTab := table();</td>
</tr>
<tr>
<td>#fill up the table</td>
</tr>
<tr>
<td>for i from 1 to 13 do</td>
</tr>
<tr>
<td>valueTab[i] := i^2;</td>
</tr>
<tr>
<td>end do;</td>
</tr>
<tr>
<td>#Produce list as result</td>
</tr>
<tr>
<td>L := convert(valueTab,list);</td>
</tr>
<tr>
<td>printf(&quot;L is: %a&quot;, L);</td>
</tr>
</tbody>
</table>

```
#initialize table
valueTab := table();

#fill up the table
for i from 1 to 13 do
    valueTab[i] := i^2;
end do;

#Produce list as result
L := convert(valueTab,list);
printf("L is: %a", L);
```

```
table([ ])
1
4
9
16
25
36
49
64
```
The nice thing about this loop is that it is no more work for us to create a list with 100 items 1, 4, 9, ... 10,000. All we have to do is change the "13".

However, there is an even shorter way to create this list. Maple has a function seq whose syntax borrows heavily from for. The seq function creates a sequence as indicated by the values of the stepping variable. To get a list, we convert the sequence into a list, as explained in convert (page 141).

**Making a list**

**Making a list with seq**

```maple
#initialize table

#Create a sequence
s := seq(i^2, i=1..13);
#Convert it to a list
L := [s];
printf("L is: %a", L);
```

```
L is: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169]
```

With this kind of facility, it is easy to quickly solve many problems that are solved with Maple functions that take lists.

**Creating a list of values**

**Creating a list of values with seq**

Create a list of the approximations to $\sqrt{1}$, $\sqrt{2}$, ... $\sqrt{20}$.

```maple
seq(evalf(sqrt(i)), i=1..20)
```

```
```

The general forms of seq are

```
seq( expression involving steppingVariable, steppingVariable=range)
```

`seq( expression involving steppingVariable, steppingVariable=range, stepValue)`. The second form is behaves like the **for... from... to ... by ...do...end do** described in the previous chapter.

**Giving lists to Maple functions**

**Giving lists to Maple functions using seq with a step value**

Create a list of the approximations to $\sqrt{2}$, $\sqrt{2^2}$, ... $\sqrt{50}$. 

```
seq(evalf(sqrt(i^2)), i=1..50, 10)
```

```
```
Giving lists to Maple functions using `seq` with a step value

\[ \text{seq}(evalf(\sqrt(i)), i = 2..5, 1) \]

\[
[1.414213562, 1.449137675, 1.483239697, 1.516375089, 1.549193338, 1.581138330, 1.612451550, 1.643167673, \\
1.67332053, 1.702983637, 1.732050808, 1.760681686, 1.788854382, 1.816590212, 1.843908891, 1.870828693, \\
1.897366596, 1.923583406, 1.949358869, 1.974841766, 2.000000000, 2.024845673, 2.049390153, 2.073644135, \\
2.09761796, 2.121320344, 2.144761059, 2.167948339, 2.190892030, 2.213594362, 2.236067977] \]  

Adding together the elements of two lists is another common operation that has a built-in shortcut.

**Adding with**

Create a list of values of \( \sin\left(\frac{n\pi}{10}\right) \) (approximately).

\[
Lsin := \text{evalf}\left(\text{seq}\left(\sin\left(\frac{i\cdot\pi}{10}\right), i = 1..10\right)\right)
\]

\[
[0.3090169944, 0.5877852524, 0.8090169944, 0.9510565165, 1., 0.9510565165, 0.8090169944, 0.5877852524, \\
0.3090169944, 0.] \]  

Create a similar list of values for \( \cos\left(\frac{n\pi}{10}\right) \)

\[
Lcos := \text{evalf}\left(\text{seq}\left(\cos\left(\frac{i\cdot\pi}{10}\right), i = 1..10\right)\right)
\]

\[
[0.9510565163, 0.8090169943, 0.5877852522, 0.3090169938, 0., -0.3090169938, -0.5877852522, -0.8090169943, \\
-0.9510565163, -1.] \]  

Add them together using "+".

\[
Lsum := Lsin + Lcos
\]

\[
[1.260073511, 1.396802247, 1.396802247, 1.260073511, 1., 0.6420395227, 0.2212317422, -0.2212317419, \\
-0.6420395219, -1.] \]  

We don't get component-wise "automatic multiplication" with "*". Maple doesn't give an error, but it doesn't do the multiplication, either.

\[
Lwrong := Lsin \cdot Lcos
\]

\[
[0.3090169944, 0.5877852524, 0.8090169944, 0.9510565165, 1., 0.9510565165, 0.8090169944, 0.5877852524, \\
0.3090169944, 0.] \]

We get this done without writing a loop to do it with the use of `seq`.

\[
Lright := \text{seq}(\text{evalf}(\text{Lsin[i]} \cdot \text{Lcos[i]}), i = 1 \ldots \text{nops(Lsin))}
\]

\[
[0.2938926262, 0.4755282582, 0.4755282581, 0.2938926257, 0., -0.2938926257, -0.4755282581, -0.4755282582, \\
-0.2938926262, -0.] \]  

"Automatic addition" works with numbers, expressions, lists, sequences, ranges, and equations, but does not work with tables, plots, or sets. "Automatic multiplication" only works with numbers and expressions. Maple does have additional data containers with use with linear algebra, vectors, and matrices for which both addition and multiplication work -- but that should wait until you need to do linear algebra.
12.3 More implicit looping with sum

Maple has another function which will use looping when it is expedient: summation. This is available textually through the \texttt{sum} operation, and through the Expression Palette as \( \sum_{i=k}^{N} f_i \). The general form of \texttt{sum} is \texttt{sum(expression, indexVariable=range)}. When the range is numerical and contains a manageable number of values, Maple will automatically do the summation through a loop.

\begin{verbatim}
 restart;
 L := [3.6, 3.9, 2.1, 3.9, 2.1, 0.7, 1.3, 2.8, 3.6, 3.7, 2.9];
 total := sum(L[i],i=1..nops(L));
\end{verbatim}

Automatic summation

\begin{verbatim}
\sum_{i=1}^{10} i \quad \#\text{This does a summation using the Expression Palette}

\sum_{i=1}^{10} \sqrt{i} \quad \#\text{This does summation using the textual form of the operation}
\end{verbatim}

However, there is more to the \texttt{sum} function than just an abbreviation of a \texttt{for} loop. One can give non-numeric value as part of the range, and Maple will use mathematical analysis to give a formula for the summation if it can. Furthermore, it will this analysis to provide a short-cut to the looping if it turns out to be faster.

\texttt{sum}\texttt{behavior}

\begin{verbatim}
\sum_{i=1}^{n} i^4 \quad \text{Maple knows the formula for the sum of 4th powers of integers.}
\end{verbatim}

\begin{verbatim}
\frac{1}{2} (n + 1)^5 - \frac{1}{2} (n + 1)^4 + \frac{1}{3} (n + 1)^3 - \frac{1}{30} n - \frac{1}{30}
\end{verbatim}

If we did the following with a \texttt{for} loop, it would take literally centuries for even a supercomputer to complete. Rather than do that, Maple figures out the formula for the sum of squares and plugs in \( n=10^{20} \).

\begin{verbatim}
\sum_{i=1}^{10^{20}} i^2
\end{verbatim}

One annoying aspect of using \texttt{sum} is that it expects the index variable to be unassigned (have no value). If you discover that you’ve used the index variable previously in the session, then you have to unassign it. This is similar to the situation that arose in (Brian, provide hyperlink) when \texttt{solve} broke because the variable being solved for had been previously assigned a value.

\texttt{sum} and previously assigned values of the index variable

\begin{verbatim}
 restart;
 L := [3.6, 3.9, 2.1, 3.9, 2.1, 0.7, 1.3, 2.8, 3.6, 3.7, 2.9];
 total := sum(L[i],i=1..nops(L));
\end{verbatim}
avg := total/nops(L); 
for i from 1 to 10 do 
  printf("Grade = %f.2, deviation= %f.2 \n",L[i], L[i]-avg); 
end do; 
stdev := sum((L[i]-avg)^2,i=1..nops(L))/avg;

[36, 39, 21, 39, 21, 07, 13, 28, 36, 37, 29] 
30.6 
2.781818182 

The error message comes when we try to compute the standard deviation. While the use of $i$ in the computation of total had no problems, the use of $i$ in the computation of the standard deviation occurred after it had been assigned values during the for loop. The way to fix this is to either use another variable that is known not to have been assigned a value (e.g. $k$), or to unassign $i$ before using it in the second sum.

restart;
L := [3.6, 3.9, 2.1, 3.9, 2.1, 0.7, 1.3, 2.8, 3.6, 3.7, 2.9];

#Sum all grades, compute average 
total := sum(L[i],i=1..nops(L)); 
avg := total/nops(L): 

#Print a table of grades and the deviation from the average for i from 1 to 10 do 
  printf("Grade = %f.2, deviation= %f.2 \n",L[i], L[i]-avg); 
end do;
i := 'i';  #Make sure i has no value before you use it in a sum. 
stdev := sum((L[i]-avg)^2,i=1..nops(L))/avg:
printf("Mean = %f.2, Standard deviation=%f.2", avg, stdev); 

[36, 39, 21, 39, 21, 07, 13, 28, 36, 37, 29] 
30.6 
2.781818182
Grade = 3.600000.2, deviation= 0.818182.2
Grade = 3.900000.2, deviation= 1.118182.2
Grade = 2.100000.2, deviation= -0.681818.2
Grade = 3.900000.2, deviation= 1.118182.2
Grade = 2.100000.2, deviation= -0.681818.2
Grade = 0.700000.2, deviation= -2.081818.2
Grade = 1.300000.2, deviation= -1.481818.2
Grade = 2.800000.2, deviation= 0.018182.2
Grade = 3.600000.2, deviation= 0.818182.2
Grade = 3.700000.2, deviation= 0.918182.2

Mean = 2.781818.2, Standard deviation=4.369935.2

12.4 Conditional repetition through while... do... end do

The general form of the while statement
while condition do
    loop body;
end do;

The line with the while contains a continuation condition which is usually an inequality such as \( x < 1 \) or \( \text{abs}(y-z) > \text{tol} \), but must at any rate be an expression that can be evaluated to be true or false. 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. The condition is evaluated only once per repetition, at the start before any of the code in the loop body is executed. If the condition is true, then the entire loop body is executed. The condition is evaluated each time Maple decides whether to execute the loop body.

Repeating condition small

Repeating while a condition is true: accumulate terms of a sum until next term is very small.

The code region below computes the sum
\[ x + \frac{3}{3!} + \frac{5}{5!} + \frac{7}{7!} + \ldots \]
until it finds a term that is tiny (less than the value of the parameter tol). It does this through repeated addition to the accumulation variable s. The continuation condition is to repeat the code within the loop body as long as the value of \( \frac{x}{i!} > \text{tol} \). This condition is evaluated only once per repetition, just before all the code in the loop body is executed. The particular term calculated during a repetition is \( (-1)^{i-1} \cdot \frac{x}{i!} \) for \( i = 1, 3, 5, \ldots \).

Some of you who have taken calculus may recognize this sum as an approximation to \( \sin(x) \). Programmers often use sums or similar means of approximations to calculate numerical values of math functions such as \( \sin \).

```
# Initialize script parameters
#tol is the size of the stopping criterion.
tol := 10e-7;
#x is the value of \( \sin(x) \) that we are trying to approximate
x := .3;

#Initialize accumulation variables
i := 1;  #i will be 1, 3, 5, 7, 9 as the repetition continues
```
term := x;
s := term;  # s accumulates the sum, we initialize it to the first term.

# Add on successive terms
while abs(term) >= tol do
  i := i+2;
  term := (-1)^((i-1)/2)*(x^i)/i!;
  s := s+term;
  printf("Term is %e, sum is %f.\n", term, s);
end do: # Suppress execution trace inside the loop (but printf still work).

# Print some summary messages.
printf("Sum is: %e compared to %e.\n", s, sin(x));
printf("Stopped when i=%d, term=%e.\n", i, term);

0.000000
  0.3
    1
      0.3
        0.3

Term is -4.500000e-03, sum is 0.295500.
Term is 2.025000e-05, sum is 0.295520.
Term is -4.339286e-08, sum is 0.295520.
Sum is: 2.955202e-01 compared to 2.955202e-01.
Stopped when i=7, term=-4.339286e-08.

Notes
The terms being computed are sometimes positive and sometimes negative. This is why the continuation condition uses the absolute value of the term rather than just the value of the term. It says 'do another repetition while the term is bigger than tol. The last iteration of the loop computes a term that's ~4.33928571410^-8. After that, the condition is not true, so the repetition stops. The final result agrees to ten decimal places with Maple's built-in sin function's.

The textual form of inequalities such as "<" and ">" are easy since there is a symbol on the keyboard for them. The following table describes the textual version of inequalities such as \( \leq \). The symbols for inequalities are sometimes referred to as relational operators.

Textual entry of relational operators

<table>
<thead>
<tr>
<th>Mathematical operator</th>
<th>Name</th>
<th>Textual entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>&lt;</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>&gt;</td>
</tr>
<tr>
<td>=</td>
<td>Equals</td>
<td>=</td>
</tr>
<tr>
<td>≠</td>
<td>Not equals</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td>≤</td>
<td>Less than or equals</td>
<td>&lt;=</td>
</tr>
<tr>
<td>≥</td>
<td>Greater than or equals</td>
<td>&gt;=</td>
</tr>
</tbody>
</table>

Another example of accumulation

Another example of accumulation
This example is similar to the previous one. It computes successive terms of a sum for the natural logarithm:

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

but stops when the term becomes smaller than the tolerance.

```plaintext
# Computation of logarithms through sums.
# Initialize script parameters
# tol is the size of the stopping criterion.
tol := 10e-4;
# x is the value of sin(x) that we are trying to approximate
x := .2;
xm1 := x-1.0:    # "x minus 1", since we use it a lot

# Initialize accumulation variables
i := 1;  # i will be 1, 2, 3, 4, 5, .. as the repetition continues
term := xm1;
s := term;  # s accumulates the sum, we initialize it to the first term.
xm1power := xm1: # to be used for powers of x minus 1, initially just xm1

printf("x is %f, tol is %e\n",x,tol);
# print out all values on one line as a check before we start.

# add on successive terms
while abs(term)>=tol do
  i := i+1;  # compute next value of the denominator
  xm1power := -xm1power*xm1; # Next power of x-1, adjusted by +- sign.
term := xm1power/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("sum is: %e compared to: %e", s, ln(x));
print(i, "terms used");
```

```
 0.0010
 0.2
 1
-0.8
-0.8
```
The approximation does not produce ten decimal places of agreement with the logarithm as calculated by the Maple library, but it's closing in on that when the repetition stop. This is because \( \text{tol} \) is set to \( 10^{-4} \), so the repetition stops as soon as a term is computed that is less than that in absolute value. If we really wanted ten places of accuracy, we might want to set \( \text{tol} \) to \( 10^{-10} \). It would make the repetition go on longer, but would produce more decimal places of agreement. It would be naive to presume that just because we set \( \text{tol} \) to \( 10^{-10} \) that the result will agree with the correct result to 10 decimal places. (What actually happens? Try it.) Mathematical analysis can be used to provide guarantees or proofs that things will work out. The area of mathematics that studies this is called numerical analysis. Many undergraduate mathematics or computer science students interested in devising efficient accurate programs for numerical computation take a numerical analysis course as a junior or senior. Such courses are

### 12.5 Patterns of looping

We have introduced quite a lot of machinery. It can be easy for beginners to be uncertain about the connection between intent (what you want to achieve) and syntax (how you say it in the programming language). Here as a short table summarizing the connections we have introduced so far.

#### Patterns of looping

<table>
<thead>
<tr>
<th>Patterns of looping</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intent</td>
<td></td>
</tr>
</tbody>
</table>
| Repeat exactly the same thing \( n \) times | to 5 do 
  print("I will not teach others to fly"); 
  end do; |
|                     | "I will not teach others to fly" |
|                     | "I will not teach others to fly" |
|                     | "I will not teach others to fly" |
|                     | "I will not teach others to fly" |
|                     | "I will not teach others to fly" |
The action within the loop depends on the value of variable stepping from an initial to a final value, adding a fixed amount to the stepper variable between each step:

\[
\text{for } t \text{ from 0 by .1 to 1 do}
\text{printf("arccos(\(f\)) = \(f\)\n", } t, \text{ arccos}(t)) \]
\text{end do;}

\[
\begin{align*}
\text{arccos}(0.000000) &= 1.570796 \\
\text{arccos}(0.100000) &= 1.470629 \\
\text{arccos}(0.200000) &= 1.369438 \\
\text{arccos}(0.300000) &= 1.266104 \\
\text{arccos}(0.400000) &= 1.159279 \\
\text{arccos}(0.500000) &= 1.047198 \\
\text{arccos}(0.600000) &= 0.927295 \\
\text{arccos}(0.700000) &= 0.795399 \\
\text{arccos}(0.800000) &= 0.643501 \\
\text{arccos}(0.900000) &= 0.451027 \\
\text{arccos}(1.000000) &= 0.000000
\end{align*}
\]

Accumulation: Variables within the loop accumulate results. Each repetition incrementally changes onto the partial result. When the repetition is finished, the entire result has been computed.

\[
x := .3: \quad \#\text{Parameter}
\]
\[
\text{prod} := 1: \quad \#\text{Accumulation variable}
\]
\[
\text{#"sum" and "product" are reserved Maple names}
\text{#So we should use something else like "s" or "prod"}
\text{#for variables.}
\]
\[
\text{#Compute the product of (x+2)*(x+3)*(x+4)*...(x+15)}
\text{for } i \text{ by 2 to 15 do}
\text{prod := (x+i)*prod;}
\text{end do:} \quad \#\text{Suppress execution trace inside the loop}
\]
\[
\text{print(prod);}
\]

\[
3.549445283 \times 10^6
\]

The loop is repeated while a continuation condition is true. (The loop stops when the continuation condition is no longer true.)

This computes an approximation to

\[
e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots
\]

\[
\text{#script parameters}
x := .2:
\text{tol} := 10^(-6):
\]
\[
\text{#accumulation variables, initialized}
i := 1:
s := 1: \quad \#\text{First term of sum}
term := 1: \quad \#\text{Next term of sum}
\]
\[
\text{while } (\text{abs(term)}\geq\text{tol}) \text{ do}
\quad \text{term := (x^i)/(i!);}
\quad \text{s := s + term;}
\quad i := i+1;
\text{end do:}
\]
\[
\text{printf("compare %e to %e\n", s, exp(x));}
\]
12.6 Combining for and while

Maple's syntax allows the use of an stepper variable and a continuation condition (other than that of the stepper variable reaching a final value.

```
for stepperVariable from initialValue to finalValue by stepValue while condition do .... end do
```

promotes a repetition which stops when *either* the stepperVariable reaches the *finalValue*, or the *condition* becomes false. If the *to* *initialValue* is omitted, then the while *condition* is the sole criterion used to decide whether to continue the repetition. This can abbreviate the coding in many standard cases.

**Examplesmall**

Example of for and while repetition: accumulate terms of a sum until next term is very small.

The code region below computes the sum

\[ x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \ldots \]

until it finds a term that is tiny (less than the value of the parameter *tol*). Compared to the previous example (Brian, please hyperlink to previous *sin* example), the main difference is that we can skip the initialization and increment of the stepper variable *i* since the *for* loop handles it for us automatically. Since there is no *finalValue*, the loop will not stop because the stepper variable reached a final value. However, at the start of each repetition, the *condition* \( \text{abs}(\text{term}) \geq \text{tol} \) will also be checked. This is what stops the repetition after a few trips through the loop.

```maple
# Initialize script parameters
#tol is the size of the stopping criterion.
tol := 10e-7;
#x is the value of sin(x) that we are trying to approximate
x := .3;

#Initialize accumulation variables
term := x;
s := term;  #s accumulates the sum, we initialize it to the first term.

#add on successive terms
for i from 1 by 2 while abs(term) >= tol do
  i := i+2;
  term := (-1)^(i-1)/i! * (x^i)/i!;
  printf("term is \%e\n", term);
  s := s+term;
end do;

#Print out the final sum, and compare it to what it's supposed to be.
printf("sum is: \%e compared to \%e.", s, sin(x));
```

0.0000010

0.3

0.3

0.3
3
-0.00450000000

term is -4.500000e-03

0.295500000

7

-4.33928610^8

term is -4.339286e-08

0.2954999566

sum is: 2.955000e-01 compared to 2.955202e-01.

We can get more decimal places of agreement between our approximation and the reference result by changing the value of \( tol \). This will cause more repetitions of the loop.

This kind of repetition is often used in practice in situations where there is no built-in function that will compute the result for you. You'd want to ask a numerical analyst to provide assurances that it's going to work well!

12.7 Troubleshooting while loops: loops that don't stop

We've already reviewed some of the basic problems with loops that come from forgetting to put in a \texttt{end do} or \texttt{do}, or putting too many in. These typically give some kind of Error, message and were discussed here. (Brian: provide hyperlink to previous section where this happens).

More pernicious are loops which do not give error messages but either stop with the wrong answer, or do not stop.

You can tell that this is happening because you are either flooded with a lot of output, or the status window in the lower left hand corner of the worksheet says \texttt{Evaluating}.... but never seems to return to \texttt{Ready}.

A runaway "infinite" loop

\begin{verbatim}
x := 3.0;
while (x<10) do
    print("sin(x)", sin(x));
end do;
\end{verbatim}

3.0 "sin(x)", 0.141120081 "sin(x)", 0.1411200081 "sin(x)", 0.1411200081 "sin(x)", 0.1411200081
etc., etc.

The problem with this loop is that it will never stop. The continuation condition is \( x<10 \), which is true initially. Because the loop body never changes the value of \( x \), it will continue to be true for every successive repetition. Presumably the programmer made a mistake and forgot to put in lines of code into the loop body that changed the value of \( x \) and (with enough repetition) made it eventually grow to 10 or larger.

In theory, you can stop long-running computations such as infinite loops with the "red stop hand" found on the Maple tool bar. Clicking on the red stop hand will eventually stop the output, but not until after a long, long wait while all the output that resulted from before you used the hand scrolls by. This is because Maple typically generates thousands of repetitions of the loop in less than a second. It takes a long time for it to print all the output from the repetitions, and you won't be able shut off the printing with the red hand.
So in practice, if you have a runaway loop you will have to force-quit your Maple session and recover the most recent version of your worksheet through File -> Recent Backups-> Recover Backup, as described in chapter 2.

Interrupting an evaluation with the "red stop hand"

After entering an equation to solve, we have realized that we really don't want to see millions of roots listed, assuming that Maple can find them all. So we click on the red hand (circled in green).

After a significant pause, the message "computation interrupted" appears and we can enter another computation for Maple to perform. There is no result from the interrupted computation, so no result is displayed. The hand becomes gray instead of red when execution is over.

This technique does not work well if the long running computation prints a lot, as might occur in a runaway loop with a verbose execution trace.

12.8 Troubleshooting loops: loops that give the wrong answer

Loops that return the wrong answer

Loops that give the wrong answer need to be handled analytically -- you need to figure out what should be happening and why. Then compare what should be happening with what the execution trace is saying does happen to diagnose your problem. "Sick loops" take a fair bit of thought to work through, since you are looking a code that gets repeatedly executed, possibly many times.

Some tips for handling loops include: a) Develop incrementally. Set your loops up for a few repetitions (two or three) initially. Verify that things work before you try to make the repetitions go on longer.

b) Use print/printf statements within the loop to make it clearer what is going on inside the repeated instructions.

Another problem with a loop is that the values provided for its control cause a different effect than you expected. Discover the difference between what the rules say will happen and what you thought would happen by observing the execution trace, possibly putting in print statements.

Finally, in accumulation loops, the initial value of the accumulator variable may be wrong, or the accumulation is not happening as planned. The control of the loop may stop the accumulation too early, or too late.

Troubleshooting loops: a script that doesn't work
s := 0;
x := .2;
Approximate exp(x) = 1 + x + x^2/2! + x^3/3! + ...

for i to 10 from 1 by 1 do
  We changed the order of to and from, but that's not an error
  s := s + x^i/i!;
end do;
printf("%f compared to expected %f", s, exp(x));

0.221403 compared to expected 1.221403

The execution trace seems to indicate that the right number of terms is being computed, but the answer we computed by our own
programming is different than what Maple's built-in exp function produces. The computed answer seems to be one less than the
answer we expect. This is different than the other examples previously shown where the programmed results agreed quite closely
with Maple's built-in computation.

Re-running the script with more information generated by the execution trace

s := 0;
x := .2;
Approximate exp(x) = 1 + x + x^2/2! + x^3/3! + ...

for i to 10 from 1 by 1 do
  printf("i= %d, x = %f, s = %f, x^i/i! = %f\n", i, x, s, x^i/i!);
  s := s + x^i/i!;
end do;
printf("%e compared to expected %e", s, exp(x));

We enhance the execution trace by having it print out the values of all the variables with each repetition. It's work to enter in this
printf statement, but it pays off in helping us find the mistake more quickly. Comparing the actual value of s with what the program
comment tells us to expect term by term, we see that while s starts off at zero in the first iteration, the first term \( \frac{x}{0!} \) is .2, not 1 as we expect. Looking at the second and third terms, we realize that we have forgotten to add in the first term to s.

There are two ways to do this -- either to start at \( i=0 \) (since \( \frac{x}{0!} \) is 1) or to keep the loop as it is and just initialize s to 1 instead of zero before the loop starts. We opt for the latter since it doesn't require remembering facts from high school days such as that 0!=1.

**Re-running the script with repairs**

| s := 1;  
x := .2;  
#Approximate exp(x) = 1 + x + x^2/2! + x^3/3! + ...  
for i to 10 from 1 by 1 do  
    printf("i= %d, x = %f, s = %f, x^i/i! = %f\n", i, x, s, x^i/i!);  
    s := s + x^i/i!;  
end do;  
printf("%e compared to expected %e", s, exp(x)); |

| i= 1, x = 0.200000, s = 1.000000, x^i/i! = 0.200000 |
| 1 |
| i= 2, x = 0.200000, s = 1.200000, x^i/i! = 0.020000 |
| 1.2 |
| i= 3, x = 0.200000, s = 1.220000, x^i/i! = 0.001333 |
| 1.2200000 |
| i= 4, x = 0.200000, s = 1.221333, x^i/i! = 0.000067 |
| 1.22133333 |
| i= 5, x = 0.200000, s = 1.221400, x^i/i! = 0.000003 |
| 1.22140000 |
| i= 6, x = 0.200000, s = 1.221403, x^i/i! = 0.000000 |
| 1.22140267 |
| i= 7, x = 0.200000, s = 1.221403, x^i/i! = 0.000000 |
| 1.221402756 |
| i= 8, x = 0.200000, s = 1.221403, x^i/i! = 0.000000 |
| 1.221402759 |
| i= 9, x = 0.200000, s = 1.221403, x^i/i! = 0.000000 |
| 1.221402759 |
| i= 10, x = 0.200000, s = 1.221403, x^i/i! = 0.000000 |
| 1.221402759 |
Making this change produces the expected agreement. At this point, we would "comment out" the printf that we put in to enhance the execution trace, and could change the value of x to compute an approximation to exp(x) for other values of x.

### 12.9 Summary of Chapter 12

<table>
<thead>
<tr>
<th>Implicit iteration</th>
<th>Example</th>
</tr>
</thead>
</table>
| Using seq and sum  | #Set parameters of script  
n := 5;  
#Unassign i just in case it's been used in the session  
#previously  
i := 'i';  
#compute consecutive powers of 10, in a list.  
tList := [seq(10^i, i=1..n)];  

#Compute logarithms of the powers you have  
yList := [seq( evalf(log(tList[i])), i=1..n) ];  

#Do a data plot  
plot(tList, yList, style=point, symbol=diamond, color=red,  
   labels=["x","log(x)"], symbolsize=30);  

#Add together the logarithms  
s := sum(yList[i], i=1..n);  
printf("Sum of logarithms is:%e\n", s);  

 5  
i  

[10, 100, 1000, 10000, 100000]  

[2.302585093, 4.605170186, 6.907755279, 9.210340372, 11.51292546]
Sum of logarithms is: 3.453878e+01.

Automatic addition

\[
L := \begin{bmatrix} 0.3, 0.5 \end{bmatrix};
\]
\[
\text{velocity} := 0.1;
\]
\[
\text{angle} := \text{convert}(55, \text{units}, \text{degrees}, \text{radians});
\]
\[
\text{dv} := \text{evalf}([\text{velocity}\times\cos(\text{angle}), \text{velocity}\times\sin(\text{angle})]);
\]

# add dv onto L
\[
\text{newL} := L + dv;
\]

Examples of while loops

Example of while...do

\[0.3, 0.5\]

0.1

\[
\frac{11}{36} \pi
\]

[0.05735764361, 0.08191520445]

[0.3573576436, 0.5819152044]

# Add on successive terms
while abs(term) >= tol do
  i := i+2;
  term := (-1)^((i-1)/2)*(x^i)/i!;
  s := s+term;
  printf("Term is %e, sum is %f.\n", term, s);
end do: # Suppress execution trace inside the loop (but printfs still work).

# Print some summary messages.
Example of for...to...do

```
for i to 10 do
    printf("i= %d, ", i);
end do;
```

```
i= 1, i= 2, i= 3, i= 4, i= 5, i= 6,
i= 7, i= 8, i= 9, i= 10,
```

Example of for...to...by...do

```
for i to 10 by 2 do
    printf("i= %d, ", i);
end do;
```

```
i= 1, i= 3, i= 5, i= 7, i= 9,
```

Example of for...to...from...by...do

```
for i to 10 from 5 by 2 do
    printf("i= %d, ", i);
end do;
```

```
i= 5, i= 7, i= 9,
```

Troubleshooting tips

Print out values of variables using execution trace

```
s := 1;
x := .2;

#Approximate exp(x) = 1 + x + x^2/2! + x^3/3! + ...

for i to 10 from 1 by 1 do
    printf("i= %d, x = %f, s = %f, x^i/i! = %f\n", i, x, s,
           x^i/i!);
    s := s + x^i/i!;
end do;

printf("%f compared to expected %f", s, exp(x));
```

Have the foresight to avoid running an infinite loop. If despite your best efforts, this happens and hitting the "red stop hand" doesn't work fast enough, kill the Maple session and start again. Hopefully you have saved your work or can use the File->Open Recent->Recover Backup.
13 Conditional Execution

13.1 Chapter Overview

We introduce the ability to selectively execute a segment of code, or an alternative segment instead through `if...then....else...end if`. When there are more than two alternatives, use `if...then...elif....elif....else...end if`.

Loops and conditional execution are found in most programming languages in a similar form. Learning how to control execution with them is a skill that usually transfers easily, with practice and opportunity, to working with many other computer tools.

13.2 Choosing alternatives: `if...then...end if`, `if... then... else...end if`

The `if` statement allows a programmer to get the computer to choose among alternative actions based on whether a condition is true or false.

The general form of the if statement is: `if condition then` code to executed if the `condition` is true

`else`

code to be executed if the `condition` is false

`end if`

**If-then-else-end if : one possible outcome**

```plaintext
grade := 3.7:  #This value will cause execution of the "then" code
if (grade >= 3.5)
    then
        printf("An excellent grade: %.2f\n", grade);
    else
        printf("A Grade: %.2f\n", grade);
    end if;

An excellent grade: 3.70
```

**If-then-else-end if : another possible outcome**

```plaintext
grade := 3.2:  #This value will cause execution of the "else" code
if (grade >= 3.5)
    then
        printf("An excellent grade: %.2f\n", grade);
    else
        printf("A grade: %.2f\n", grade);
    end if;

A grade: 3.20
```

Obviously, if you are planning to do something once, you can probably just look at what's being computed and decide yourself what to do. The value of `if` is when you plan to use a script many times, or if you need to make many condi-
tional decisions within a loop in a single execution of a script. If is helpful because you won't have to stop to think about what happens each time. You can just set things up ahead of time so that in either case the right thing happens.

If-then-else-end in a loop

```maple
#Initialize a list with some numbers, some grade point
# averages
L := [3.6, 3.9, 2.1, 3.9, 2.1, 0.7, 1.3, 2.8, 3.6, 3.7, 2.9];

#Initialize some accumulation variables. We will use them
#to count how many grades are 3.5 or above, and
#how many are below.
countHigh := 0;
countLow := 0;

#This goes first, before a loop, as a table header
printf("Grades at least 3.5:\n");

#i steps through the possible index values for L.
#Recall that nops(L) is the length of the list.
for i from 1 to nops(L) do
    if (L[i] >= 3.5)
        printf("%.2f\n", L[i]); #print out the grade with 2 decimals
        countHigh := countHigh+1; #add one to the "high" counter
    else
        countLow := countLow+1; #add one to the "low" counter
    end if;
end do:

printf("There were %d grades at or above, and %d grades below 3.5.",countHigh, countLow);
```

Grades at least 3.5:
3.60
3.90
3.90
3.60
3.70

There were 5 grades at or above, and 6 grades below 3.5.

A fairly common occurrence is having the else alternative specify no actions at all. Rather then have if...then...else end if, Maple allows you to say just if...then...end if.
In University of West Dakota at Hoople, the Dean's list is determined by computing the average of all student's grade point averages, and selecting those whose gpa is at least .5 point higher than the average.

```
restart; #unassign any previous values
L := [3.6, 3.9, 2.1, 3.9, 2.1, 0.7, 1.3, 2.8, 3.6, 3.7, 3.8, 2.0]:
deanCount := 0:

#Compute the average for all students by
#using sum to add up all the grades, and dividing by the
#number of grades
avg := sum(L[i],i=1..nops(L))/nops(L):

#Print out table header.
printf("Grade average of %d students is: %f.\n", nops(L), avg);
printf("Grades at least one point above average (Dean's List):\n");

for i from 1 to nops(L) do
    if (L[i] >= (avg+1) )
    then
        printf("%.2f\n", L[i]);
        deanCount := deanCount+1;
    end if;
end do:
printf("\nThere were %d students making the Dean's List.",deanCount);
```

Grade average of 12 students is: 2.791667.
Grades at least one point above average (Dean's List):
3.90
3.90
3.80

There were 3 students making the Dean's List.

13.3 Choosing among cases with If-then-elif-elif...else-end if

The if statements we have seen so far can be described as being useful when:

19. The condition was used to select between two alternative series of actions.

20. The condition was used to select between one series, and doing nothing as the alternative.

Another situation that arises in practice is to select between several (more than two) alternatives.

**if-then-elif-else-endif**

<table>
<thead>
<tr>
<th>General form</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (condition 1) then statements separated by ; or : elif (condition 2) then statements separated by ; or : (more alternatives specified by more elifs)</td>
<td>If condition 1 is true when the statement is executed, then the first series of statements is executed. Otherwise, if condition 1 is not true but condition 2 is true, then the second series as executed.</td>
</tr>
</tbody>
</table>
One way to view this is that you are describing different cases, depending on what is true. The *else* portion of the statement specifies what is to be done if none of the previous cases apply.

### Example

**Example of If-then-elif-elif-...-else-end if**

```plaintext
# Initialize L to be a list of grades
L := [3.6, 3.9, 0.9, 2.1, 3.9, 2.1, 0.7, 1.3, 2.8, 3.6, 3.7, 2.9, 1.3];

# Initialize counters for the number of As, Bs, etc. (silently)
countA := 0;
countB := 0;
countC := 0;
countD := 0;
countF := 0;

# Process each element of the list, with the following cases:  
# As (3.3 and above)  
# Bs (at or above 2.7 and below an A)  
# Cs (at or above 1.7 and)  
# Ds (at or above 1.0 and below a C)  
# Fs (below 1.0).
for i from 1 to nops(L) do
  x := L[i]; # Use "x" instead of typing L[i] several times
  if (x >= 3.3) then
    countA := countA +1;
  elif (x >=2.7) # this is tried only if first condition false
    then
      countB := countB +1;
  elif (x > =1.7) 
    then
      countC := countC +1;
  elif (x >=1.0)
    then
      countD := countD +1;
  else # we will be here only if everything else hasn't worked
    else  
      countF := countF +1;
  end if;
end do:
```
printf("\nThere were %d As, %d Bs, %d Cs, %d Ds, and %d Fs.", countA,countB,countC, countD,countF);

There were 5 As, 2 Bs, 2 Cs, 2 Ds, and 2 Fs.

It is possible to include if statements as one of the actions inside another if statement. This is useful where a case may have subcases. We will discuss this situation in more detail in later chapters.

13.4 For the curious: comparing control statements in conventional programming languages

Almost all conventional programming languages (C, Mathematica, C#, Java, Matlab, Maple, C++, NXT, Python, Perl, Visual Basic, etc.) have the "loop" idea and the "selection/case" ideas in them. The main difference in how they specify them through the use of different words or symbols. Java for example, uses { and } rather than "then" or "end if" to delimit the begin and end of segments of actions.

Portionconditional

Portion of a Java program with a conditional

```java
int countThree, countSubThree; int [] L= {1,5,9,2,6,7,-2,1,7}; // Set up counters countThree=0; countSubThree=0; // Loop through L counting how many are above three for (int i=0; i<L.length; i++) { if (L[i]>3) { countThree++;} else {countSubThree++;} }
System.out.println("three and above:"+countThree+ " below three: "+ countSubThree);
```

While this portion of the Java program is quite intelligible to someone who is more experienced in Maple, a full program to do this would require additional statements that we don't should here. Note that the printing statement `println` uses "+" not for numerical addition but to combine the output into a single string that `println` can handle. Getting used to these idiosyncrasies are part of the overhead of working with more than one tool on a computer.

The Matlab program follow with a loop and a conditional may look similar to a Maple program, although it uses different functions to extract size ("size" rather than "nops") and there is different syntax (no do, just "end", "=" instead of from and ":=" instead of to).

Portionconditional

Portion of a Matlab M-file with a loop and a conditional

```matlab
%Program to count how many elements of a vector/list are greater than and less than three countThree=0; countSubThree=0; L=[1,5,9,2,6,7,-2,1,7]; [r,c]=size(L); %Loop for i from 1 to c, where c is the number of items in L (length of L) for i=1:c if L(i)>3 countThree=countThree+1; else countSubThree=countSubThree+1; end; end; %Use formatted printing to print out things. sprintf(\'Three and Above: \%d, Below three: \%d\', [countThree, countSubThree])
```

There may be other ways of doing the same task in a language that may be easier in that particular language. For example, summing together numbers in Java involve writing a loop that gathers the sum of the numbers, whereas in Maple one could use "sum" and get the job done in one line. C, Java and C++ have a "?" operator which one could use to write the if of the above example more succinctly even though it would work the same as the kind we have worked about. No language will ever be "the most convenient" for all tasks, which is one of the reasons why so many endure.

However, the main message of this section is that it knowledge of how to set up loops and conditional execution applies to many languages. Learning how to do it in Maple transfers to most other languages you may choose to work with in the future.
13.5 Troubleshooting if statements

While, for, and if statements have multiple requirements -- you have to have words such as "while", "do", "if", "else", etc. written in exactly the right sequence or else the language processor will reject them. Sixty years of advances in programming language technology haven't changed the situation much. Unfortunately the only sure cure for this is to gather enough experience with making such mistakes, getting expert help to fix them initially, and then remembering enough from the experience to be able to avoid such mistakes in the future or at the very least be able to diagnose them more quickly. If there is no expert help around, then you could try to get a working example from a book to work within your Maple document, and then incrementally modify it so that it continues to work and yet becomes similar to what you really want.

Error

<table>
<thead>
<tr>
<th>Error messages if statements</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example</td>
<td></td>
</tr>
<tr>
<td>x := 3;</td>
<td>No missing pieces here.</td>
</tr>
<tr>
<td>countThree := 0;</td>
<td></td>
</tr>
<tr>
<td>countSubThree := 0;</td>
<td></td>
</tr>
<tr>
<td>if x&gt;3</td>
<td></td>
</tr>
<tr>
<td>then</td>
<td></td>
</tr>
<tr>
<td>countThree := countThree +1;</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>countSubThree := countSubThree +1;</td>
<td></td>
</tr>
<tr>
<td>end if;</td>
<td></td>
</tr>
<tr>
<td>printf(&quot;three and above:%d, below three: %d&quot;, countThree, countSubThree);</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>three and above:0, below three: 1</td>
<td></td>
</tr>
</tbody>
</table>

x := 3; countThree := 0; countSubThree := 0;
if x>3
    countThree := countThree +1;
else
    countSubThree := countSubThree +1;
end if;
printf("three and above:%d, below three: %d", countThree, countSubThree);
3
0
0
1
three and above:0, below three: 1

We forgot to put in the word "then". After the error, the cursor is positioned in the midst of the line after the "if", which is the first time that Maple's input processor notices that there's something wrong. To handle this error, one looks backwards from the position of the cursor to see if there's anything missing. Admittedly there's not a lot of information given that the cause is a missing then, but this is fairly typical in working with programming languages. You'll get an indication that there's something wrong, and the point at which the wrongness was first noticed, but you won't be able to depend on the error message to tell you exactly what was wrong. If computer technology could do that reliably, systems would just fix the errors automatically. But that would probably require artificial intelligence far beyond the state of the art.
We forgot to end the statement with end if (or fi).

```plaintext
x := 3;
countThree := 0;
countSubThree := 0;
if x>3
then
countThree := countThree +1;
else
countSubThree := countSubThree +1;
printf("three and above:%d, below three: %d",
countThree, countSubThree);
end do;
printf("three and above:%d, below three: %d",
countThree, countSubThree);
```

We said "end do" rather than "end if".

```plaintext
x := 3;
countThree := 0;
countSubThree := 0;
if x>3
then
countThree := countThree +1;
else
countSubThree := countSubThree +1;
end do;
printf("three and above:%d, below three: %d",
countThree, countSubThree);
```

Actually, this is not a mistake. Evidently "end if" and "end" are acceptable here. "fi" is another acceptable way of ending an if statement.
countSubThree := countSubThree +1;
end;

printf("three and above:%d, below three: %d",
   countThree, countSubThree);

3
0
0
1

three and above:0, below three: 1

x := 3;
countThree := 0;
countSubThree := 0;
if x>3
   then
      countThree := countThree +1;
   else
      countSubThree := countSubThree +1;
   end if

printf("three and above:%d, below three: %d",
   countThree, countSubThree);

3
0
0

Error, missing operator or `;`

x := 3;
countThree := 0;
countSubThree := 0;
IF x>3
THEN
   countThree := countThree +1;
ELSE
   countSubThree := countSubThree +1;
END IF;

printf("three and above:%d, below three: %d",
   countThree, countSubThree);

3
0

We get the same error message as in the second item in this example. However, this time the problem is that we left off the semi-colon after the end if. We need it if there’s an instruction after the if statement in the script.

Programmers need to the consider that there could be more than one kind of mistake that would provoke an error message. This kind of situation can be encountered with almost any sort of programming system (e.g. C, Matlab, Java, etc.) due the limitations of conventional language-processing technology.

The mistake here is that IF does not mean the same thing as “if” to Maple -- case matters. Executing this script gives an error message with the cursor positioned right after the IF. The error message is saying that Maple doesn’t know what to do with the IF.
### 13.6 Chapter summary

<table>
<thead>
<tr>
<th>General form</th>
<th>Explanation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if condition</code>&lt;br&gt;<code>then action; .... action;</code>&lt;br&gt;<code>end if</code></td>
<td>If <code>condition</code> is true when the statement is executed, then the actions are performed, otherwise they aren't. <code>fi</code> can be used instead of <code>end if</code>. They mean the same thing.</td>
<td><code>if (x &gt;= 3.5)</code>&lt;br&gt;<code>then</code>&lt;br&gt;<code>printf(&quot;%.2f\n&quot;, x);</code>&lt;br&gt;<code>countHigh := countHigh+1;</code>&lt;br&gt;<code>end if;</code></td>
</tr>
<tr>
<td><code>if condition</code>&lt;br&gt;<code>then action; .... action; else other action; .... other action;</code>&lt;br&gt;<code>end if</code></td>
<td>If <code>condition</code> is true when the statement is executed, then the actions are performed, otherwise the alternative actions are performed. <code>fi</code> can be used instead of <code>end if</code>. They mean the same thing.</td>
<td><code>if (x &gt;= 3.5)</code>&lt;br&gt;<code>then</code>&lt;br&gt;<code>printf(&quot;%.2f\n&quot;, x);</code>&lt;br&gt;<code>countHigh := countHigh+1;</code>&lt;br&gt;<code>else</code>&lt;br&gt;<code>countLow := countLow+1;</code>&lt;br&gt;<code>end if;</code></td>
</tr>
<tr>
<td><code>if condition1</code>&lt;br&gt;<code>then action; .... action; elsif condition2 actions2; .... actions2; elsif condition3 actions3; .... actions3;</code>&lt;br&gt;<code>end if</code></td>
<td>If <code>condition1</code> is true when the statement is executed, then the actions are performed. If not, then <code>condition2</code> is true, then the second group of actions is performed, etc. If none of the conditions are true, then nothing is done. This is like a &quot;case by case&quot; description of what to do, where the various conditions describe the cases.</td>
<td><code># Leaves weather variable unchanged except for two cases where there's change.</code>&lt;br&gt;<code>if barometer=&quot;falling&quot; and weather=&quot;It's sunny&quot;</code>&lt;br&gt;<code>then</code>&lt;br&gt;<code>weather := &quot;It's cloudy&quot;;</code>&lt;br&gt;<code>elsif</code>&lt;br&gt;<code>barometer=&quot;rising&quot; and weather=&quot;It's cloudy&quot;</code>&lt;br&gt;<code>then</code>&lt;br&gt;<code>weather=&quot;It's sunny&quot;;</code>&lt;br&gt;<code>end if;</code></td>
</tr>
<tr>
<td><code>if condition1</code>&lt;br&gt;<code>then action; .... action; elsif condition2 actions2; .... actions2; elsif condition3 actions3; .... actions3; else actions otherwise; .... actions otherwise;</code>&lt;br&gt;<code>end if</code></td>
<td>Like the previous situation, only there is an additional set of actions that are done if none of the conditions are true. This is like a &quot;case by case&quot; description of what to do, where the various conditions describe the cases, and the last series of actions describe what happens otherwise if none of the other cases hold.</td>
<td><code>#Determine the location of an (x,y) point.</code>&lt;br&gt;<code># We assume that x and y already have values from assignments made in the previous point in execution.</code>&lt;br&gt;<code>if x&gt;=0 and y&gt;=0</code>&lt;br&gt;<code>then</code>&lt;br&gt;<code>location := &quot;quadrant 1&quot;;</code>&lt;br&gt;<code>elsif</code>&lt;br&gt;<code>x&gt;=0 and y&lt;0</code>&lt;br&gt;<code>then</code>&lt;br&gt;<code>location := &quot;quadrant 2&quot;;</code>&lt;br&gt;<code>elsif x&lt;0 and y&gt;=0</code>&lt;br&gt;<code>then</code>&lt;br&gt;<code>location := &quot;quadrant 4&quot;;</code>&lt;br&gt;<code>else</code>&lt;br&gt;<code>location := &quot;quadrant 3&quot;;</code>&lt;br&gt;<code>end if;</code></td>
</tr>
</tbody>
</table>
location := "unknown";
end if;
Index

Symbols
! (exclamation point factorial), 10
# (program comment), 120
%d, 116, 134
%f, 116, 134
(, 9
), 9
- (minus or dash), 9
; (semi-colon), 120
=, 113, 120
:=, 39
; (semi-colon), 120
< (less than), 167
<= (less than or equals), 167
<default>, 146, 167, 172
= (equals), 167
= (equation), 17
> (greater than), 167
>= (greater than or equals), 167
^ (caret), 10

A
abs(a), 66
accumulation, 170, 173
actual parameter, 55
acute accent marks, 59
adding lists, 163
addition (+), 9
algebraic unknown, 29
Algol 68 (programming language), 146
Alpha, 67
alpha, 67
altitude, 106
animate, 110
  function, 104
animation of a moving object, 104
animation speed, 105
animation tool bar, 105
animations
  exporting to GIF format, 112
apostrophe, 87
apostrophes, 59
arc cosine, 21
arc sine, 21
arc tangent, 21
argument, 55
arguments
  of function, 69
arithmetic, 8
  approximate, 22
  calculator-style, 30
  exact, 11
  operations, 9
arrow, 72
assignment, 18, 39
  of a plot structure, 98
assistant
  Unit Converter, 90
asterisk, 9
autosaving, 16
average
  numbers in a list, 64
Axiom, 5

B
backquotes, 59
backspace key, 13
backup, 134, 173
backups
  retrieving, 16
ball, 106
binomial, 66
Blackberry, 64
blue, 101
brackets, 41
by, 148

C
C, 183
C (the programming language), 5–6, 119, 183
C# (the programming language), 183
C++ (the programming language), 5, 183
Canadian roots, 101
caret (power), 10
CHI, 67
chi, 67
circle, 101, 103, 108
  function, 101
circle-X, 7
clickable interface, 55
code edit region, 115
coding, 123
colon, 97, 113, 120
Colorado, 42
colors
  plotting, 63
comma, 41
command-=, 38
command-C (copy operation), 13, 44
command-e (execute region), 120
command-L, 38
command-R, 37–38
command-T, 37
command-V (paste operation), 13, 44
command-X (cut operation), 13
comment
  program (#), 120
comments
  header, 121
  program, 115
completion
  command, 70
composition
  of functions, 84
computing
  algebraic, 1
  graphical, 1
  high performance, 3
  logical, 1
  numerical, 1
constant
  symbolic, 29
containers
  data, 139
continuation condition, 166
control-=, 30, 38
control-C (copy operation), 13, 44
control-click, 17
control-e (execute region), 120
control-L, 38
control-R, 37–38
control-T, 37–38
control-V (paste operation), 13, 44
control-X (cut operation), 13
convert, 86, 106, 141
  sequence into list, 162
convert a list into a sequence, 64, 68
convert a list into a set, 64, 68
convert a list into a string, 64, 68
copy (editing operation), 13
copy and paste, 44
crash
  Maple, 16
curly braces, 41
currency, 86
CurveFitting, 97
cut (editing operation), 13
decimal point, 29
delimeters in Maple: ( ) [ ] { }, 12
digits, 22
display, 98, 110
  function, 99, 101
divide and conquer, 45
division (/), 9
do, 145
documentation
  need for, 2
dot
  centered, 10
double-quotes, 59
down the road
  just, 64
E
  e
    symbolic constant, 29
Edit->Execute->Selection, 44
eelif, 181
ellipse, 103
else, 179, 181
empty list, 41
end do, 145
end if, 179, 181
ENGR101, 45
enter, 44
enter (key), 37, 66
entries, 141, 143
equals, 167
equation, 17, 139
  heating, 126
error
  (in sum) summation variable previously assigned, 165
  expecting 1 argument, 60
  invalid color specification, 59
  invalid fraction, 13
  invalid left hand side in assignment, 142
  invalid matrix/vector product, 12
  invalid operator parameter name, 75
  invalid power, 56
  invalid sequence, 12
  invalid sum/difference, 11
  logic, 15
  missing operation, 12
  missing operator or ‘;’, 185
  numeric exception: division by zero, 12
  rounding, 31
  unable to match delimiters, 12, 56, 60
  unexpected option, 59–60
  vocabulary, 15
error message, 11

D
daisy chaining
  functions, 84
dashed rectangle, 7
data structures, 139
days, 87
escape key (ESC), 70
Euro, 84
eval, 67, 77
evalf, 78
evaluate, 38
   at a point, 44
evaluate and display in-line (control-=), 30
evaluate at point, 42
evaluating
   preventing, 87
   state, 16
evaluating a formula, 102
Evaluating..., 172
evaluation, 66
do of expressions, 8
execute region, 120
execute worksheet, 66
execution trace, 151
exponential function (exp), 66–67
exponentiation, 14
expression, 139
expressions
   algebraic, 17
   from palette, 20

F
factor, 17, 77
factorial (!), 10
false, 166
fence post error, 23
File->New->Document, 45
final value, 146
find something similar, 45
floating point number, 29
for, 146, 171
formatted output (%a
%f
, %d, etc.), 117
Formatting
   output
      as currency, 86
Fortran, 5
fraction, 29
freeze frame
   animation, 105
from, 148
fruit flies, 72
fsolve, 76, 139
ft, 86
function
   idea of, 69
function definition, 72, 75
function definition alternative, 76
functions
   define, 72
designing from word problems, 81
thousands of built-in, 70
fuzzy notions, 45

G
gallon, 87
gallons
   Imperial, 64
GAMMA, 67
gamma, 67
gigaflot, 3
graphical user interface (GUI), 4
Graphics Exchange Format (GIF), 112
greater than, 167
greater than or equals, 167
Greek letters, 67
green, 101

H
header comments, 121
help
   on-line, 60, 67, 71
   quick, 7
   using on-line, 72
horsepower, 87
hour, 87
hypotenuse, 92

I
i
   imaginary, 67
   imaginary number, 30, 33
Ideal Gas Law, 81
if, 179, 181
immutable, 139
inch, 86
index, 40, 101, 140, 166
index variable, 146
indices, 140, 143
infinite loop, 158
infinity, 29, 43, 67
initial value, 146
Insert->2-D Math, 37
Insert->Label, 38
Insert->Text, 37
integer, 37
interactive, 5
interpreted system, 66
interrupting Maple, 172
inverse trig functions, 21
J
Java, 5
Java (the programming language), 183
justification of formatted output, 116

K
Kessel Run, 87
key, 140
kilometer, 88
kilopascals, 82
kilowatt, 87
km, 86

L
label, 9, 38
labels for axes
plotting, 63
LeastSquares, 97
left arrow key, 13
left hand side (right-click menu), 23, 26
less than, 167
less than or equals, 167
lhs, 77, 139
limited-precision number, 29
line
function, 100–101
linear algebra, 12, 163
list, 41, 57, 139
add together items from, 64
count number of items in, 64
empty, 53, 139
one item from, 64
operations on, 64
part of, 64
ln, 30
logarithm
base 10, 66
base b, 66
base e, 21
logic error, 15, 23
loop, 145
loop body, 145
looping, 145
Los Alamos National Laboratory, 3

M
Macsyma, 5
Maple, 4, 183
as a word processor, 37
document mode, 7
Maple 13, 4
match parentheses, 45
math entry mode, 37
Mathematica, 5–6, 183
mathematical model, 1
mathematical models, 102
Matlab, 5, 34, 119, 183
Symbolic Toolkit, 6
matrices, 163
maxdigits, 14
mean, 166
meter, 86
micrometer, 87
mile, 86
millimeter, 86
minus, 9
minute, 87
mistake
typographical
how to fix, 13
mu, 67
multiple systems, 6
multiplication (*), 9

N
National Forest Service, 43
negation (-), 9
nops, 139–140
not equals, 167
NULL, 115–116
NULL (the empty sequence), 53
numbers
exact, 29–30
floating point, 29–30
limited precision, 30
limited-precision, 29
whole, 29
NXT (the programming language), 183

O
Oak Ridge National Laboratory, 3
Octave, 5
omega, 67
on-line documentation, 60
op, 139
opening worksheet, 66
orange, 101
oscilloscope, 45

P
packages, 97
palette, 20
Common Symbols, 29–30, 59, 66
Expression, 29, 55, 57, 66, 69, 72–74, 76, 164
Greek, 66
square root, 21
parameter, 44
parameterized plots, 103
parentheses, 9, 55
parsec, 87
paste (editing operation), 13
patterns of looping, 169
Perl (the programming language), 183
petaflop, 3
Pi, 67
symbolic constant, 29, 32
pick names, 44
pitfalls
  common
    in Maple, 69
playing
  animation, 105
plot, 77, 139
  builder, 60
colors, 59
labels, 59
options, 63
structure, 98
PlotBuilder, 20
plots
  package, 99, 104
plotting, 18
plotting a function, 102
plottool, 100
plottools, 100
plus, 9
pop-up, 75
power (^), 10
precision
  fixed, 31
pressure, 91, 98
preventing
  evaluation, 87
print, 115
printf, 116
program code, 123
Python, 5–6, 34
Python (the programming language), 183
rectangle
  dashed (entry), 7
red, 101
red stop hand, 172
refactor
  code, 85
  relational operators, 167
repeat n times, 169
replay
  animation, 106
retrieving
  a Maple session, 15
return (key), 37
reuse
  of software, 2
rhs, 77, 139
right hand side, 42
right hand side (of equation), 22
right hand side (right-click menu), 23, 26
right-click, 17
right-click->solve, 33
right-click->solve->numerically, 33
rounding, 31

S
Sage, 5
Saving
  a Maple document, 15
scaling=constrained, 103
scientific notation, 30
scripting, 42
search field
  for help, 60
select
  entry, 40
select entry (right-click menu), 23, 26
semi-colon, 120
seq, 162
sequence, 12, 41, 92, 139
  empty, 53, 139
set, 41, 140
  empty, 53, 140
sheep, 42
side effect, 115
simulation, 1
single-quotes, 59
slash, 9
small steps, 45
Solo
  Han, 87
solve, 40, 55, 67, 76, 139
  a system of equations, 65
  by right-clicking or control-clicking, 17
numerical, 22
solving an equation, 102
sprintf, 117
square root
from palette, 21
standard deviation, 165
Star Wars, 87
state
of variables in worksheet, 66
state display
of Maple, 16
statement
program, 120
stepper variable, 146
stepping, 170
stop playing
animation, 105
string, 116, 134
sublist, 64
subscript, 38
subtraction (-), 9
sum, 164, 166
supercomputer, 3
symbolsize, 108

table, 140
tan, 30
Tau, 67
tau, 67
temperature, 91, 98
template
textual, 70
teraflop, 3
terminating
Maple computation, 172
test, 45
text entry mode, 37
textual entry, 55
then, 179, 181
to, 145
tol, 166
tool bar
animation, 105
Tools->Spellcheck, 38
trace
execution, 128
triangle, 92
troubleshooting, 45
function definitions, 75
if statements, 184
loops, 155
tables, 142
textual input, 67
unit conversion, 88
true, 166

U
unassign, 39, 45
underscore, 38
Unit Converter assistant, 90
units, 106
Units Calculator, 64
Use of printf, 116, 134

V
Vancouver stock exchange, 34
vectors, 163
version
number, 133
vertical range, 56
Visual Basic (the programming language), 183
visualization, 1

W
Warning
premature end of input use <Shift> + <Enter> to avoid
this message, 146
whattype, 118
while, 166, 170–171
widgets, 84
with, 97
worksheet
appearance, 66
state, 66

Y
yard, 87

Z
zeroes
leading or trailing, 116