Chapter 8 Evaluation, display of results, and functions

This chapter is primarily conceptual, as opposed to giving you a lot of new functions that you can do calculations with. It's time to explain a bit more about the ideas that explain what's going on when you operate Maple, and how functions know what kinds of things they're working with.

Section 8.1 Evaluation of expressions and read-eval-print

The Maple document interface works in a standard way for everything you do. First it accepts the characters you type until you type `enter` (called `return` on some keyboards). Then it processes (evaluates) what you typed, often performing calculations. Finally it displays the results, and awaits the next input. This is referred to as the read-eval-print loop or read-eval-print cycle. Most computer languages that provide "instant line at a time response" (Maple, Scheme, Python, Perl, Matlab, etc.) behave in this read-eval-print fashion. This is what happens when you operate a Maple worksheet:

In Maple, read-eval-print operates in the following way:

1. The expression you type is scanned for meaning. If there are things that are not grammatical for the Maple language, then a message is printed out and no further processing of the input occurs. The user is expected to type in a corrected line, or use mouse/keyboard editing of what was typed, to try again. This is the read phase.
2. The expression is evaluated. Operations or function invocations are performed as specified. Assignments (:=) are performed if requested. If expression asks for impossible things (e.g. to divide by zero), then run-time error messages will be printed, and processing stops. The user is then free to edit the error-generating input and try again. This is the evaluation phase.
3. The result (assuming that there have been no errors in the read and eval phases) is printed in a nicely formatted way. This is the print phase.

The worksheet will allow you to perform more rounds of reading/evaluation/printing once the first round is complete.

In Maple, if you end a line of input with a colon (:), then it suppresses the print phase. Other systems have different syntax for suppressing the print phase, but you can expect there to be some way of doing this.

### Example 8.1 Suppressing display of results

<table>
<thead>
<tr>
<th>( r ) := 10!</th>
<th>( \begin{array}{c} 3628800 \ (1.1.1) \end{array} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s ) := 100!;</td>
<td></td>
</tr>
</tbody>
</table>

Ordinarily results of assignments are displayed.

This would be a big number, but because we end the expression with a : it suppresses the display of the answer. The number is assigned to \( s \), though.

We see that \( s \) must have the value of whatever 100! is, because if we divided it by 99!, we get the leftovers of the quotient as a result.
$p := \text{plot}\left(\{\sin(2t), \sin(t) \cdot 2\}, t = 0 \ldots 2\pi\right)$

We get a plot data structure do a plot, and that is what is assigned to $p$. We see an abbreviated form of the result (in blue).

$p2 := \text{plot}\left(\{\sec(2t), \sec(t) \cdot 2\}, t = 0 \ldots \frac{\pi}{6}\right)$:

$p2$ is assigned a plot structure, but we don't see the PLOT(...) because we ended the input with a colon.

When we evaluate $p2$ (with a colon) the "print" part of the read-eval-print operation of the worksheet is suppressed.

When we evaluate $p2$ (without a colon), the plot data structure's information is printed, as part of the normal read-eval-print operation of the worksheet.
If you have a very long running computation, clicking on the "red stop hand" on the Maple tool bar attempts to interrupt the computation. Since the operation was interrupted in the "eval" part of the read-eval-print cycle, only the message "Computation interrupted" is printed.

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.

Section 8.2 Variations on inputs to and outputs from functions

Section 8.2.1 Types
One of the basic concepts in many programming languages is the notion of type. In most computer languages, every value that the language can treat belongs to a distinct type. Some of the types are numerical -- integer, rational number, floating point and are found in many different computer languages. Other types in Maple -- equations, symbols (names), expressions, ranges, lists, sets -- are in Maple because of its technical orientation but may not be built-in types in other languages.

In addition to being a kind of value, a type also allows certain kinds of permissible implemented operations. Everyone is aware that the standard arithmetic operations +*/^-^, abs, sin, sqrt, etc. work on numbers, but not all of them make sense for lists, sets, plot structures. The operations rhs and lhs work on equations but not on numbers or expressions. Sometimes you will see error messages resulting from trying to apply an operation to a value
whose type does not allow that operation.

Some type names are symbols such as integer or float. Others are names of a form that we have not seen before -- "backquoted names". Here is an example of a backquoted name: `=`. Backquoted names can be typed from the keyboard, but you need to type the "acute accent" kind of apostrophe `, not the ordinary kind of apostrophe ', or the double-quote symbol ". ' and " are also used in Maple, but for other purposes, so you can't use them interchangeably with ` in Maple.

You can find out what type Maple thinks a value is through the whattype function. The whattype function will take a single parameter, which can be of any time. It will return a name that is the type of the parameter's value.

<table>
<thead>
<tr>
<th>Example 8.2.1.1 whattype and types in Maple</th>
</tr>
</thead>
<tbody>
<tr>
<td>whattype(3)          integer          (1.2.1.1)</td>
</tr>
<tr>
<td>whattype(3.5)        float            (1.2.1.2)</td>
</tr>
<tr>
<td>whattype\left( \frac{3}{5} \right)     fraction        (1.2.1.3)</td>
</tr>
<tr>
<td>whattype(x = y + 1)  <code>=</code>             (1.2.1.4)</td>
</tr>
<tr>
<td>whattype(3 .. 5)     ..               (1.2.1.5)</td>
</tr>
<tr>
<td>whattype(x + y)      <code>+</code>             (1.2.1.6)</td>
</tr>
<tr>
<td>whattype(x - y)      <code>+</code>             (1.2.1.7)</td>
</tr>
<tr>
<td>whattype(x \cdot y)  <code>\ast</code>           (1.2.1.8)</td>
</tr>
<tr>
<td>whattype\left( \frac{x}{y + 1} \right) <code>\ast</code>   (1.2.1.9)</td>
</tr>
<tr>
<td>whattype\left( y^{-2} \right)          <code>\wedge</code>          (1.2.1.10)</td>
</tr>
<tr>
<td>whattype\left( \frac{1}{y} \right)     <code>\wedge</code>          (1.2.1.11)</td>
</tr>
<tr>
<td>whattype\left( {a, b, {1, 2}} \right) set          (1.2.1.12)</td>
</tr>
</tbody>
</table>

Some of the official names of types in Maple are words. Others are expressions such as `=` or `\ast`. The name of the type for equations is `=`. A sum is of type `+` (referred to as "sum"). An expression that's a difference is also of the same type. Maple is thinking of this as the sum of x and -y. "\ast" is referred to as "product". A quotient of algebraic expressions is also a product.
| `whattype(x)` | symbol | (1.2.1.13) | Maple is thinking of \( \frac{1}{y} \) as the same as \( y^{-1} \). |
| `whattype( (x) → sin(\(\frac{x}{2}\))` | procedure | (1.2.1.14) |
| `whattype(sin)` | symbol | (1.2.1.15) |
| `whattype(solve)` | symbol | (1.2.1.16) |
| `whattype(sin(x))` | function | (1.2.1.17) |
| `whattype(plot(sin(x), x = 0 .. 1))` | function | (1.2.1.18) |
| `w := [1, a, x + 1]` | (1.2.1.19) |
| `whattype(w)` | list | (1.2.1.20) |
| `whattype(whattype(w))` | symbol | (1.2.1.21) |
| `whattype(whattype(x·y))` | symbol | (1.2.1.22) |

As we will see in later chapters, functions defined by -> are referred to as "procedures" in Maple.

Note that if a symbol is assigned a value, `whattype` (like most other functions) looks up the value of the symbol and returns as a result of that.

To Maple, the result of calculating `sin(x)` (for a symbol `x`), and the result of doing a plot are the same type: `function`.

Evidently the result of `whattype(w)` (e.g. `list`) is just another symbol to Maple.

This result shows that `*` is a symbol for Maple as well.

Expressions of type `*`, `+`, and `^` are referred to collectively as *algebraic*.

The reason why computer languages have types is that all values belonging to a particular type (e.g. `integer`, `list`, or `equation`) are implemented in the computer in a similar way. This means that the processing of all values of that type can be described in a unified way. This makes it feasible to build systems that can (in theory at least) be expected to process all integers, or all equations, or any kind of list.

Maple has a lot more built-in types than some common programming languages such as C or C++, but considerably fewer than other languages such as Java or C#. Those languages may have thousands of types. Fortunately, no one ever has to deal with all of the types at once. For a particular task, one determines the functions that are needed to get the job done, and then figures out what types of values those functions need. In reading about the functions in the online documentation, one usually gets enough information about the types that are needed to do the work.

Some Maple operations such as `+` (addition) or `*` (multiplication) allow you to mix numbers of varying types together. However, others will give you an error because the operation...
<table>
<thead>
<tr>
<th>Example 8.2.1.2 Mixed types and restrictions in types in Maple functions and operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>3.5</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>5 mod 3</td>
</tr>
</tbody>
</table>

| 2.0                                                                         | Division (as well as addition, subtraction, multiplication, etc.) allows you to add an integer and a floating point number together. The type of the result is a float. |
| 3                                                                            | Dividing two integers, however produces a rational number -- an exact result? How did Maple decide to do that? By looking at the types of the operands of the operation. |
|                                                                             | Here the types are mixed, so a floating point result is mandated. |
| 2.0                                                                          | modp is a function that returns the remainder of the first number divided by the second number, which both must be integers. It is sometimes referred to as the "modular function". |
| 3                                                                            | Maple has an alternative syntax for modp. |
|                                                                             | Note that this hardly makes sense, so there's an error. |
| 5 mod 3                                                                     | Rather than pontificate about whether the value of the first argument is, Maple just looked at its type (which was float) and gave an error. It wants both arguments to be of integer type. Even though 5.0 is mathematically an integer, the artifact stored in the computer is not of that type. |

**Section 8.2.2 Characterizing the behavior of functions by parameter/result type and number of parameters**

In programming languages, functions can be categorized according to **what type of results they return**, **what type of value their parameters must be**, and **the number of parameters they use**.

Most of the functions we have seen in Maple so far, e.g. `abs`, `sin`, `ln` take as input one parameter which if numeric (e.g. `integer`, `rational`, or `float`) takes the function to return a numerical result. These functions however, if given a symbol or expression, often return an expression back which **is the same expression**. We have also seen functions that take two or more numbers as
arguments and return a numerical result. However, we have also seen and written functions that take lists or sets as inputs, and return similar things as their result, such as `sort`. The basic requirement about Maple functions is that parameters can be of any Maple type except a `sequence`: numbers, lists, sets, plot structures, ranges, equations, and so on. Maple functions can return any type of thing including a sequence. This kind of type/parameter freedom is common in most programming languages, and experienced programmers find the extra flexibility to be worth the additional complexity in conceptual understanding.

<table>
<thead>
<tr>
<th>Function</th>
<th>Number of parameters</th>
<th>Types of parameters</th>
<th>Type of result</th>
<th>Example and/or anti-example</th>
</tr>
</thead>
<tbody>
<tr>
<td>nops</td>
<td>1</td>
<td>various</td>
<td>integer</td>
<td><code>nops(3 = x - 1)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>nops([a, b, c, d])</code></td>
</tr>
<tr>
<td>rhs</td>
<td>1</td>
<td>equation</td>
<td>various</td>
<td><code>rhs(3 = x - 1)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>rhs(\frac{3}{5})</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Error, invalid input: <code>rhs</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>received 3/5, which is not</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>valid for its 1st argument,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>expr</code></td>
</tr>
<tr>
<td>trunc</td>
<td>1</td>
<td>A numerical type</td>
<td>integer</td>
<td><code>trunc(\frac{3}{5})</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>0</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>trunc(1.1)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>1</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>trunc(x + y)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>trunc(x + y)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>trunc(3 = x - 1)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Error, invalid input: <code>\texttt{\textbackslash simp1/trunc}</code> expects its 1st argument, al, to be of type algebraic, but received 3 = x-1</td>
</tr>
<tr>
<td>binomial</td>
<td>2</td>
<td>integer, integer</td>
<td>integer</td>
<td><code>binomial(3, 5)</code></td>
</tr>
</tbody>
</table>
Section 8.2.3  Examples of user-defined functions that take various types and
numbers of parameters and return various types of results

Functions often take more than one parameter if more than one piece of information needs to
be supplied in order to do the computation. An alternative to creating a function with multiple

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort</td>
<td>list, ordering directive</td>
<td>list</td>
<td><code>sort([3, 1, 5])</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>[1, 3, 5]</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>sort([3, 1, 5], '&gt;</code>)`</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>[5, 3, 1]</code></td>
</tr>
<tr>
<td>solve</td>
<td>equation or algebraic, symbol</td>
<td>numeric or algebraic or a sequence</td>
<td><code>solve(3 = x - 1, x)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>4</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>solve(y = \frac{1}{x - 1}, x)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>\frac{y + 1}{y}</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>solve(x^2 - 1, x)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>1, -1</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>solve(\{x^2 - 1 = y^2, x + y = 3\}, (x, y))</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>\{x = \frac{5}{3}, y = \frac{4}{3}\}</code></td>
</tr>
<tr>
<td>modp</td>
<td>integer, integer</td>
<td>integer</td>
<td><code>modp(5, 3)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>2</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>modp(118, 17)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>16</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>modp(118.0, 17)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Error, invalid argument for modp or mods</td>
</tr>
</tbody>
</table>
parameters is to design a function that takes a list as a single parameter. The programming of the function typically uses the selection operation \( L[i] \) to get at individual parts of the list in order to do the computation. This is one way that two or more pieces of information can be fed as input into a function.

Similarly, if more than one piece of information needs to be communicated as a result, the function can be designed to create a list, sequence, or other composite type which holds all the results. Typically the list is created with \( \text{seq} \), or \( \text{map} \) onto an existing list.

### Example 8.2.3.1 Functions that take different types and numbers of arguments

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
</table>
| \[ f1 := (n) \rightarrow \begin{cases} x & x < n \\ n & n \geq n \end{cases} \]
| \( n \rightarrow \text{piecewise} (x < n, x, n \leq x, n) \) | \( \text{(1.2.3.1)} \) We're used to a function that takes one numerical argument, or two. But \( f1 \) and \( f2 \) take numbers as inputs and produce algebraic expressions (rather than just numbers) as output. |
| \[ f2 := (n, m) \rightarrow f1(n) + f1(m) \]
| \( (n, m) \rightarrow f1(n) + f1(m) \) | \( \text{(1.2.3.2)} \) |
| \( f1(5) \)
| \( \begin{cases} x & x < 5 \\ 5 & 5 \leq x \end{cases} \) | \( \text{(1.2.3.3)} \) |
| \( f2(5, 6) \)
| \( \begin{cases} x & x < 5 \\ 5 & 5 \leq x \end{cases} + \begin{cases} x & x < 6 \\ 6 & 6 \leq x \end{cases} \) | \( \text{(1.2.3.4)} \) Note that \( f1(5) \) and \( f2(5, 6) \) are both expressions involving \( x \) and numbers. Those two expressions are what we're plotting. In other words, |
| \( \text{plot} \{ f1(5) \text{,} f2(5, 6) \}, x = 0 \ldots 10 \) | |

\[ \text{standardStats := (L) \rightarrow [nops(L)],} \]

We define a function that takes a list of
max(L), min(L), Statistics[Median](L),
Statistics[Mean](L)

L→ [nops(L), max(L), min(L),
Statistics[Median](L),
Statistics[Mean](L)]

\[QBRatings2008 := [105.5, 97.4, 96.9, 96.2, 95.0, 93.8, 92.7, 91.4, 90.2, 89.4, 87.7, 87.5, 87.0, 86.4, 86.4, 86.0, 85.4, 84.7, 84.3, 81.7, 81.0, 80.3, 80.2, 80.1, 79.6, 77.1, 76.0, 73.7, 72.6, 71.4, 70.0, 66.5] \]

\[\text{standardStats}(QBRatings2008) \]


\[\text{standardStats}(battingAvgs2008) \]

\[40, 0.364, 0.296, 0.3060000000, 0.3099750000 \]

\[\text{numbers, and returns a list} \] that consists of:

- the number of items in the list
- the largest value
- the smallest value
- the median value
- the mean value

We give it a list of quarter back ratings for the top NFL quarterbacks for the 2008 season (http://sports.espn.go.com/nfl/statistics?stat=pass&sort=rat&league=nfl&season=2&year=2008)

We see that the mean and median ratings are close, around 85.

We create another list of numbers, this time the top batting averages for Major League Baseball in 2008. (http://sports.espn.go.com/mlb/stats/batting?league=mlb)

We see that the mean average is a bit higher than the median. The people at the top were evidently much better than the rest of the pack.
We define a function that takes a list and produces a single number as a result. Each argument is itself a list of two numbers. The result is a single number which is the average of the two.

\[
avgFunc := (L) \rightarrow \frac{(L[1] + L[2])}{2}
\]

\(L \rightarrow \frac{1}{2} L_1 + \frac{1}{2} L_2\) (1.2.3.10)

\(L_1 := [5, 9]\) (1.2.3.11)

\(L_2 := [14, 16]\) (1.2.3.12)

\(avgFunc(L_1)\) (1.2.3.13)

\(dataList := [ [5, 9], [14, 16], [32, 31], [14, 19], [32, 0]]\) (1.2.3.14)

\(result := map(avgFunc, dataList)\) (1.2.3.15)

\(evalf(result)\) (1.2.3.16)

\([7, 15, \frac{63}{2}, \frac{33}{2}, 16]\) (1.2.3.17)

We define a function that takes a list of numbers as produces a range as a result.

\(hiloFunc := (L) \rightarrow \min(L) .. \max(L)\) (1.2.3.18)

\(hiloFunc(battingAvgs2008)\)

\(battingAvgs2008 .. battingAvgs2008\)

We define a function that takes a list of numbers and produces a list of lists as a result. The result is rank ordered, and each item of the result consists of a pair with the rank order and the value associated with it.

\(indexList := (L) \rightarrow \text{seq}([i, L[i]], i = 1 .. \text{nops}(L))]\) (1.2.3.19)

\(L \rightarrow \text{seq}([i, L[i]], i = 1 .. \text{nops}(L))]\)

\(rankList := (L) \rightarrow indexList(sort(L, \text{`>`}))\) (1.2.3.20)

\(L \rightarrow \text{indexList}(\text{sort}(L, \text{`>`}))\)

\(ptsNBA := [13.1, 15.8, 14.6, 17.8, 13.8, 10.5, 12.4, 10.8, 11.2, 25.9, 17.7, 17.2, 19.7, 19.8, 19.7, 10.6, 11.8, 20.1, 26.8, 14.7, 7.6, 12.3, 10.6, 20.9, 16.5, 21.1, 22.7, 19.1, 18.6, 9.4, 25.1, 20.5, 23.8, 12.7, 19.6, 23., 15.6, 18.7, 16.5, 23.7, 13.8, 14.9, 12.1, 9.1, 22.8, 21.8, 7.4, 11.9, 10.8, 10.7]:\)

We first learned how to sort in descending order in section XX.

The data is the per game NBA scoring leaders as of January 2, 2009. (http://www.nba.com/statistics/player/Scoring.jsp?league=00&season=2008&conf=OVERALL&position=0&splitType=9&splitScope=GAME&qualified=Y&yearsExp=-1&sortOrder=4&splitDD=All\%20Teams).

We use a colon to suppress printing of the
rankList uses both a function we created (indexList) and the built-in sorting function.

Recall from Section XX pointplot function needs as its first argument a list of points, where each point is a list of two numbers. The second argument to the function is an equation of the form view=[xrange, yrange] describing the extent of the horizontal and vertical axes. The third argument is a list of labels for the horizontal and vertical axes.

This plots rank against points per game. We see a somewhat linear relationship between rank and scoring, although the people at the top and bottom seem to be on lines with a different slope than the people in the middle. It would be interesting to see whether this kind of relationship holds up in other leagues... but oh wait, we can do that easily.

As before we use a colon to suppress printing of this list. It would just be an echo of what we typed in.

We got this from http://rivals.yahoo.
18.3, 18.3, 18.2, 18.2, 18.1, 18.1, 18, 18, 18, 17.8, 17.8]:

\[ \text{rankList}(\text{ptsNCAAWomens}) = \begin{bmatrix}
[1, 25.3],& [2, 24.5],& [3, 24.1],
[4, 23.8],& [5, 23.1],& [6, 23],
[7, 22.5],& [8, 22.1],& [9, 22],
[10, 21.6],& [11, 21.5],& [12, 21.4],
[13, 20.8],& [14, 20.2],& [15, 20.2],
[16, 20.1],& [17, 20.1],& [18, 20],& [19, 19.9],
[20, 19.5],& [21, 19.5],& [22, 19.4],
[23, 19.3],& [24, 19.3],& [25, 19.2],
[26, 19.1],& [27, 19],& [28, 19],& [29, 18.9],
[30, 18.9],& [31, 18.8],& [32, 18.8],
[33, 18.8],& [34, 18.6],& [35, 18.5],
[36, 18.5],& [37, 18.5],& [38, 18.4],
[39, 18.3],& [40, 18.3],& [41, 18.2],
[42, 18.2],& [43, 18.1],& [44, 18.1],
[45, 18.1],& [46, 18],& [47, 18],
[48, 18],& [49, 17.8],& [50, 17.8]
\end{bmatrix} \]

\[ \text{plots[pointplot]}((1.2.3.22), \text{view} = [1 .. \text{nops(\text{ptsNCAAWomens})}, \text{hiloFunc(\text{ptsNCAAWomens})}, \text{labels} = ["rank", "points per game"]}) \]

This list describes the points per game of the NCAA women's basketball Division I leaders, as of January 6, 2009.

We apply the functions we've already developed to this new data. Note that we did not have to adjust the programming of the function even though the length of the list is different.

We copied and pasted the previous plot command and changed it to reflect the new data. Were we to do this a few more times, we would develop a script or a function to reduce the amount of editing we have to do for each item.
Evidently the situation with NCAA top women is different than with the NBA's top men. This time there looks like leaders are broken up into only two groups, each with its distinct linear slope. This example illustrates how a decent interactive computation system allows you to do this kind of exploration casually. We'll let you form further theories about the nature of sports scoring on your own.

While it takes more getting used to, writing functions that process aggregates such as lists allows us to process differing amounts of items with no changes necessary to the script or function. For example, in the processing of scores in the above example, the functions for `standardStats` or the expression for doing the point plotting did not need to be modified when we gave the baseball scores, which had a different number of items in it, compared to the football or basketball data.

Here is another point to consider about processing aggregates of data. Suppose we have a problem where two pieces of information are needed to solve the problem. We could write a Maple solution for this by developing a function that takes two parameters as input. However, we could also write a function that takes a single argument which we expect to be a list with two values. While either one is acceptable, the latter is easier to use if we are processing an aggregation of values, because we can use `map` on a list of lists to do all the work at once.

**Example 8.2.3.2 Processing problems whose functions take multiple values as input.**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>triArea1 := (width, height) → \frac{1}{2} \cdot width \cdot height (width, height) → \frac{1}{2} width height (1.2.3.23)</code></td>
<td>We are given the width and height of a number of triangles and wish to calculate their area. Here is a &quot;traditional&quot; function for doing this.</td>
</tr>
<tr>
<td><code>triArea1(3, 5)</code></td>
<td>$\frac{15}{2} (1.2.3.24)$</td>
</tr>
<tr>
<td><code>triArea1(6.8, 3)</code></td>
<td>$10.20000000 (1.2.3.25)$</td>
</tr>
<tr>
<td><code>triAreaL := (L) → \frac{1}{2} \cdot L[1] \cdot L[2]</code></td>
<td>The result of this is of &quot;type float&quot; because multiplication produces such a result when at least one of its inputs is a float.</td>
</tr>
<tr>
<td></td>
<td>Here is an alternative way of doing this, using a single list (with two values) as the</td>
</tr>
</tbody>
</table>
\[ L \rightarrow \frac{1}{2} L_1 L_2 \]  
\( \text{triAreaL}([3, 5]) \)
\[ \frac{15}{2} \]  
\( \text{triAreaL}([6.8, 3]) \)
\[ 10.20000000 \]  
\( \text{triAreaL}(3, 5) \)
\[ \frac{1}{2} \ 3_1 \ 3_2 \)

We get gobbledygook as a result if we try to use this function the way we used the other one. Evidently it's trying to use "3" as a list. You would think that would be an error but evidently Maple just treats it as another symbolic result.

```maple
map(triAreaL, [3, 5, 6.8, 3, 47, .3])
Error, invalid input: triAreaL uses a 2nd argument, height, which is missing
```

We want to give our area-calculating function three problems to solve. But map doesn't know how to group the information within the list together in clumps of two values.

We can give triAreaL a list of lists, which provides enough structure to get the multiple problems solved. `map` applies `triAreaL` to each element of the list, and returns a list of results. Since the result of `triAreaL([3,5])`, the result of `triAreaL([6.8,3])`, etc. are each numbers, we get back a list of numbers.

```maple
map(triAreaL, [[3, 5], [6.8, 3], [47, .3]])
\[ \left[ \frac{15}{2}, 10.20000000, 7.050000000 \right] \]
```

Here we develop a function that returns a list of two results. Given a value for a radius \( r \), it computes the circumference and area of a circle with that radius.

```maple
c := (r) \rightarrow [r, 2 \cdot \pi \cdot r, \pi \cdot r^2]
c;r := (r) \rightarrow [r, 2 \pi r, \pi r^2]
```

Applying `map` to a list of radii gives a list of lists as a result.

```maple
map(c, [1, 3.0, 51/10])
\[ \left[ 1, 2 \pi, \pi \right], \left[ 3.0, 6.0 \pi, 9.00 \pi \right], \left[ \frac{51}{10}, \frac{51}{5} \pi, \frac{2601}{100} \pi \right] \]
```

`evalf` works on a list of lists as well as it does most other expressions or aggregations that have numbers in them.
### Section 8.2.4 Functions that take no arguments

Some functions take no inputs. Another way of describing this is that they take NULL (the empty sequence) as an input. The functions that do this can be described as two varieties: a) either they do the same thing each time they are invoked, or b) they refer to information kept internally by the Maple system.

Even though no information needs to be provided to such functions, you still need to put parentheses ( ) after the name of the function in order to get the computer to perform the functional action.

Although it would seem that a zero-argument function would do the same thing all the time, this is not always the case.

<table>
<thead>
<tr>
<th>Example 8.2.4.1 Functions taking zero arguments</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Example</strong></td>
<td><strong>Commentary</strong></td>
</tr>
<tr>
<td>[ \text{sameOld} := ( ) \to 42 ]</td>
<td>sameOld function that always returns 42.</td>
</tr>
<tr>
<td>[ \text{sameOld}() ]</td>
<td>This invokes the \text{sameOld} function. Its result (as always) is 42.</td>
</tr>
<tr>
<td>[ \text{plotter2} := ( ) \to \text{plot}(\sin(x), x = 0 .. 10) ]</td>
<td>plotter2 does the same plot all the time.</td>
</tr>
<tr>
<td>[ \text{plotter2}() ]</td>
<td>Typing \text{plotter2} just gives the name of the function. It does not invoke the function.</td>
</tr>
<tr>
<td></td>
<td>This invokes the function \text{plotter2} with no arguments. In other words, the () following the function name is mandatory since that is where the information about the arguments is given.</td>
</tr>
</tbody>
</table>
This is a built-in function that prints out how many seconds of computer time have been used in the current session. Note that it does not do the same thing all the time.

StringTools\texttt{[FormatTime]}( )
"2009-01-10"

It's possible for a function definition to refer to variables other than the parameters. If that is done, then if those variables have been assigned values, then the values are used.

If you are puzzled about why \texttt{(1.2.4.13)} isn't $3 \, y$, re-read section 6.1.2. If a function definition uses as a parameter name a variable that has already been assigned (such as $x$ in this situation), that assignment is ignored within the function definition. This allows one to use parameters within a function definition without having to worry
if something else has already claimed the use of that variable. In this situation, \( x \) as a parameter to \( g \) has nothing to do with the use of \( x \) in the previous assignment ((1.2.4.10)).

### Section 8.2.3 Functions with side effects

Certain kinds of functions do not have very interesting outputs, in that they always return the same thing (typically NULL, the empty sequence). However, they have useful effects because they change the state of the system, or the worksheet display.

\textit{print} is a function that can take zero or more arguments. When evaluated, it causes the results of evaluating its arguments to be printed in a nicely formatted way. \textit{lprint} is similar, except it displays the results in textual, "one dimensional" way. This can be useful if you want to cut and paste the results into something that can't handle Maple's 2 dimensional input, such as Maple TA (in 2009, at least).

\textit{printf} is "formatted" printing. The first parameter is a \textit{format string} -- a sequence of typed characters surrounded by " " (quotation marks). The subsequent parameters are expressions that will be printed according to the format specified by the string. The way that the format string describes how to print things is arcane, but powerful. This style of doing formatting is borrowed from the C programming language )see http://www.codepedia.com/1/CPrintf). Other languages such as Python and Perl also imitate C's \textit{printf}.

All of these printing functions take any number of arguments, and return NULL as a result. Thus they are used for their effect (causing results to be displayed) rather than for the final result that they compute. The printing is called a \textit{side effect} as it isn't captured in the result returned by the function.

We haven't used \textit{print} much to date because the ordinary read-eval-print cycle of operation of the worksheet already provides printing. However, in coming work we will want to print out results during the operation of a long-running function. We will also need the specially-formatted output of \textit{printf} in printing out nice-looking tables of numbers.

#### Examples 8.2.3.1 Functions with side effects: \textit{print}, and \textit{printf}

<table>
<thead>
<tr>
<th>Examples</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{restart}</td>
<td>The \textit{restart} command erases any previous assignments done in the worksheet.</td>
</tr>
<tr>
<td>( r := \text{solve}(3 \cdot x + 5 = y, x) )</td>
<td>( \frac{-5}{3} + \frac{1}{3} y ) (1.2.5.1) print displays the sequence of values of the expressions appearing as arguments and returns NULL as its result. If done interactively, the printed result is also given a</td>
</tr>
<tr>
<td>\textit{print}(r)</td>
<td>(1.2.5.2)</td>
</tr>
</tbody>
</table>
\[
\frac{-5}{3} + \frac{1}{3} y
\]  
\[\text{(1.2.5.2)}\]

\[
result := \text{print}(r, \{1, 2, 3\} \text{ union } \{1, 4, 5\})
\]
\[
\frac{-5}{3} + \frac{1}{3} y, \{1, 2, 3, 4, 5\}
\]  
\[\text{(1.2.5.3)}\]

\[
lprint(\text{temp = expand}\left((x^2 + e^{x/2})^2\right))
\]
\[
\text{temp} = 81 + 18*\text{exp}(3/2) + (\text{exp}(3/2))^2
\]

The value assigned to \textit{result} was \text{NULL}, so displaying its value produces nothing.

\textit{print} isn't too useful when you are doing interactive computing in Maple as we have been doing so far. It is much more useful when doing advanced kinds of programming, such as we will be doing before long.

\textit{lprint} works similarly to \textit{print} except that it prints things out in the style of textual input. You would get the same result if you selected a worksheet result and did Format->Convert->1D Math Input. This kind of "one dimensional" format for mathematical expressions is similar to that used in many other programming languages. Thus, \textit{lprint} can be useful for quick formatting of a small formula you've calculated in Maple but want to use in another programming language. More extensive support is provided by the CodeGeneration package (see Maple on-line help on more about this.)

\[
\text{result} := \text{printf}\left("%6.2f \ %6.2e \ %d\n%a","\right)
\]
\[
\text{evalf}(\pi), \ \text{evalf}(\pi^{1000}), 32, x + \frac{y}{2}
\]
\[
3.14 \quad 1.41e+497 \quad 32
\]
\[
3 + 1/2*y
\]  
\[\text{(1.2.5.4)}\]

The format string given to \textit{printf} says that the first value should be printed out as a six digit decimal point number with two digits after the decimal point, the second as a number in scientific notation with two digits after the decimal point, and the third as an integer. "\n" places a line break into the output display. The Last format item in the string allows for output of an arbitrary Maple expression. All the output appears in "textual" format, suitable for use as Maple textual input.

\textit{printf} or something highly similar to it are found in other technically-oriented languages, such as C and Matlab.

We will use \textit{printf} in the future when we have more need to print out tables of results in neatly organized rows and columns.
Section 8.2.4 Functions that take variable numbers of arguments

Many of the common mathematical functions require only one argument, which must be of a particular type or from a limited selection of types. For example, \( \sin \) expects only one argument, which must either be numeric or algebraic. Many useful functions have several different forms. For example, \( \text{solve} \) can take two arguments (an equation and a variable to solve for), or one (an expression involving one variable). Still others, such as \( \text{plot} \), require a minimum number of arguments (something to plot and range of values), and optionally any of a large number of other arguments (labels, colors, drawing style, etc.).

Programming languages that allow functions that take variable numbers of arguments or different types of values for the same parameter are said to allow a kind of functional polymorphism. Most computer languages developed in the past 20 years allow this, but some of the older ones (e.g. C or Fortran) are not as flexible.

Section 8.2.5 Functions in programming versus functions in mathematics

The concept of function is fundamental in both mathematics and in programming, as a way of describing something that takes inputs and produces outputs. However, the needs of programming tends to push this much further than is typically seen in algebra or calculus courses. Programmers need functions that have non-numeric parameters or return non-numeric results, functions that have many parameters, functions that have no parameters, functions that return "no result" (NULL), functions that work with different numbers of parameters each time they are invoked. The two fields use the same terminology (parameter, argument, function invocation, result) to describe functions, but then diverge in what kinds of uses they make of functions.

Another difference between mathematics and programming is how the two fields name functions. Most functions you read about in a textbook have one letter names: \( f \), \( g \), or \( h \). Those names are used over and over again in various contexts. A few famous ones, such as \( \sin \), or \( \ln \) have names that are two or three letters long. In computing, since files are being created to record work done on many different problems, there is a higher value placed on naming functions so that their purpose is easy to remember. This makes programmers give mnemonic names to the functions that they create, so that they can tell (almost) at a glance what the function is doing. It is probably better to name a function \( \text{circumferenceAndArea} \), or at least an abbreviation \( \text{circArea} \) rather than just \( f \). When you refer to the work in the future, you will be able to tell quickly what the function is up to rather than having to spend more time reading text or figuring out what formulas are doing.

Types in programming languages are similar to the concept of domains and ranges of functions in mathematics, but as we have seen the number of types in play in a programming languages is typically much larger than the number in use in elementary algebra or calculus. Furthermore, systems such as Maple typically decide what type of value a user is entering from the way it looks (e.g. 3 is an integer 3.0 is a float) rather than looking more deeply into the meaning of whatever is being entered.
Section 8.3 Random number generation

Section 8.Z Chapter summary

Summary 8.Z.1 Basic facts about functions

- Functions can have zero or more arguments, of any type except expression
- Functions can return a sequence of zero or more values of any type
- Mathematical functions typically have very short names, often one letter long. Programming functions often have longer, mnemonic names, so that it is easier to recall the activity in a worksheet when it is consulted in the future.
- A function that takes multiple arguments has a counterpart function that takes a list with multiple items. The latter is more useful if you want to apply map.

Summary 8.Z.2 Basic facts about types

- whattype(...) is a function that returns the name of the type of value given as its argument
- All values of a particular Maple type have a standard storage format. Conversely, values of different types have incompatible storage formats.
- Functions in most programming languages can accept values only of particular types. They will give an error message if a value not of that type is given to it.
- Some functions are more permissive and allow several different types to be valid forms of input.
- Computer languages such as Maple determine the type of a value from the way it looks. The division function decides whether to give a float or a rational number as its result based on the types of its input, not whether mathematically they are whole numbers or not.

Example 8.3.1 Pseudorandom numbers

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
</table>
| `rollDie := rand(1..6)`  
`proc( )`  
  `proc( )`  
    `option builtin`  
    `= RandNumberInterface;`  
  `end proc(6, 6, 3) + 1`  
`end proc`  
`rollDie( )`  
  `5` | We create a (pseudo)random generator and name it `rollDie`. `rollDie` is a function that takes zero arguments. It returns a number picked (pseudo)randomly from the range 1 to 6. We call our function once and get a number between 1 and 6. The ( ) after the name is required in order to get a (pseudo)random number. |

Example Commentary

- `rollDie` is a function that takes zero arguments. It returns a number picked (pseudo)randomly from the range 1 to 6. We call our function once and get a number between 1 and 6. The ( ) after the name is required in order to get a (pseudo)random number.
rollDie( )

2

seq(rollDie( ), i = 1 .. 10)
5, 6, 2, 3, 4, 4, 6, 5, 3, 1

rollDie

rollDie

Invoking the function again produces another random number.

Calling the function ten times gives use a random sequence of numbers between 1 and 6. Each number is equally likely.

Note that giving the name of a function but not invoking it does not produce any random numbers. rollDie, as opposed to rollDie(), does not invoke the function.

flipCoin := rand(0 .. 1)

proc( )

proc( )

option builtin

= RandNumberInterface;

end proc(6, 2, 1)

end proc

[seq(flipCoin( ), i = 1 .. 10)]

[0, 1, 1, 0, 1, 1, 1, 1, 0]

convert((1.4.7), `+`)  

7

We define a "coin flipper" a function that chooses between 0 and 1 randomly. We might be thinking of 0 as meaning heads and 1 meaning tails.

We generate a list with ten tosses.

This is another way to add together all the elements of the list. It converts the list into something of the "addition expression" type. Maple then automatically simplifies 0 + 0 + 1 + ... into a single number.

If we generate 1000 pseudorandom numbers, we see that we get tails about 49% of the time. It seems plausible that we might expect the same from really tossing a coin 1000 times.

call(1.4.7)
`+`  

0.497000000