The Maple Dictionary with Examples

by John V. Matthews, III

Contents

Chapter 1

Introduction

1.1 First things first...

It was once said that there were three things to remember about learning: Vocabulary, vocabulary and vocabulary. The point was that many of us find that the stumbling blocks to learning a subject are removed when we get a good grasp of the vocabulary for that subject. For instance, a student would find it very hard indeed to learn quantum mechanics without knowing what the words "atom", "electron", "proton" and "spin" each meant.

The purpose of this book is to educate you on the vocabulary of **Maple V** . I hope that this book will ease the process of learning **Maple V** and allow you to focus on the math that can be done with **Maple V** . If you are spared the pain of searching through the help pages or trying to find a good example of a command in the tutorials, then I have achieved my goal.

One very important thing to note is that this book is written with **Maple V** , Release 3 in mind. Many commands may work in previous releases, but there is no guarantee that they will. By the same token, the usefulness of these commands may be limited in versions after Release 3.

Direct any questions or answers to `jvmatthe@pams.ncsu.edu`.

1.2 How to use this book

In designing this book, I have tried to consider the many things that I think that students would find useful.

The second chapter is an index of basic **Maple V** symbols. This includes basic math symbols as well as some special symbols used in the **Maple V** program.

The third chapter is an index of constants that **Maple V** uses.

The fourth and most extensive chapter is an index of many of the **Maple V** commands that would be useful to the first or second semester calculus student. This index includes many examples, and tries to give an adequate description of how a command works as well as hints or warnings that a user might need to be aware of.

1.3 Special symbols used in this dictionary

For simplicity in the definitions, the following symbols will be used throughout the dictionary. When these words are italicized, they stand for a part of a definition or an argument of a command.

a, b will be real numbers, usually specifying the upper and lower limits of a range.

command will stand for a **Maple V** command.

condition will be some expression that evaluates as `true` or `false`.

deqn will stand for a differential equation.

direction will be `right` or `left` and only appears in limits.

equation will stand for an equation.

expression will stand for an expression.

i will be some indexing variable.

IC will stand for initial conditions in a differential equation.

L will stand for a list.

m, n will stand for an integer.

option will stand for extra options for a command.

polynomial will stand for a polynomial (univariate or multivariate).

relation will be something of the form $x < y$, $x > y$, $x = y$, etc.

series will stand for a **Maple V** series.

statements will stand for a single **Maple V** command or a group of commands.

subexpression will stand for some subexpression in a given expression.

type will be a **Maple V** data type.

x will stand for a variable or expression.

Chapter 2

Maple V Symbols

? The `?` is used to retrieve the help pages on a certain function or topic. If `?` is entered on a line by itself, the system returns the help page for `help`.

Example This retrieves the help page for the command `solve`.

```
> ?solve
```

Example This retrieves the help page for the `rightbox` command in the `student` package.

```
> ?student[rightbox]
```

?? The `??` command is used to find the syntax for the supplied command. It is used in the same way that `?` is used.

??? The `???` command returns examples for the given command. It is used in the same way that `?` is used.

@ The `@` symbol is used to compose two functions. It is equivalent to the $\circ$ that is used in math texts. Its meaning is shown below:

$$\texttt{(f@g)(x)} = (f \circ g)(x) = f(g(x))$$

Example

```
> f := x -> x^2;
```

$$f := x \to x^2$$

```
> g := x -> x+1;
```

$$g := x \to x + 1$$

```
> (f@g)(x);
```

$$(x+1)^2$$

@@ The `@@` symbol is used to compose a function with itself a certain number of times. A non-negative integer must follow this symbol. Its meaning is shown below:

$$\texttt{(f@@n)(x)} = (\overbrace{f \circ f \circ \cdots \circ f}^{n \text{ times}})(x) = \overbrace{f(f(f(\cdots f(x)\cdots)))}^{n \text{ times}}$$

Example

```
> f := x -> x^3;
```

$$f := x \to x^3$$

```
> (f@@5)(x);
```

$$x^{243}$$

```
> (D@@2)(f);
```

$$x \to 6x$$

```
> (D@@2)(f)(2);
```

$$12$$

! The factorial function.

Example

```
> 5!
```

$$120$$

Chapter 3

Maple V Constants

`Digits` `Digits` can be used to define the number of digits of accuracy used by **Maple V** . It is set to 10 by default. However it can be set by the user in the following way.

$\boxed{\text{Example}}$

```
> Digits;
```

$$10$$

```
> evalf(exp(1));
```

$$2.718281828$$

```
> Digits := 15;
```

$$Digits := 15$$

```
> evalf(exp(1));
```

$$2.71828182845905$$

`E` The Euler e which is roughly equivalent to 2.7182818. This is also the base for the natural logarithm. This is equivalent to `exp(1)`.

WARNING: In **Maple V** , you must use `E` and not `e` when you mean the constant e.

`I` The imaginary i which is defined to be $\sqrt{-1}$.

WARNING: In **Maple V** , you must use `I` when you mean $\sqrt{-1}$ and not `i`.

`infinity` The constant `infinity` represents the mathematical symbol ∞.

`gamma` The constant which is defined by

$$\gamma = \lim_{n \to \infty} \left(\left(\sum_{k=1}^{n} \frac{1}{k} \right) - \ln(n) \right) \approx .5772156649$$

`Pi` Everybody's favorite constant. Approximately 3.141592654.

WARNING: Maple V does not recognize `pi` or `PI` as the same thing as `Pi`. Be aware that `pi` can be used as a variable name, while `Pi` is defined as a constant and cannot be used as a variable.

Chapter 4

Maple V Functions & Commands

abs

Syntax: `abs(`*expression*`)`

The `abs` function is used to take the **abs**olute value of the given *expression*. If the expression is complex (i.e. contains `I`) then its magnitude is returned.

Example

```
> abs(5);
```

$$5$$

```
> abs(1+I);
```

$$\sqrt{2}$$

alias

Syntax: `alias(`$newname = oldname$`)`

The `alias` command allows the user to rename an object using an equation of the form $newname = oldname$ where *newname* is the new name and *oldname* is the old name. The constant `I` is an example of an alias. When the alias command is used, it replies with a list of all aliases currently known to the system.

Example

```
> f := sin(x) + sin(x)^2 + sin(x)^3;
```

$$f := \sin(x) + \sin(x)^2 + \sin(x)^3$$

```
> alias(A=sin(x));
```

$$I, A$$

Note that at this point, all current aliases are listed, including the new one `A` that we just created.

```
> f;
```

$$A + A^2 + A^3$$

```
> alias(A=A);
```

$$I$$

This last line has "unassigned" the alias.

```
> f;
```

$$f := \sin(x) + \sin(x)^2 + \sin(x)^3$$

`and` Syntax: $expression_1$ `and` $expression_2$

The `and` is a logical connector between two conditions $expression_1$ and $expression_2$ where $expression_1$ and $expression_2$ may be conditions like $a > b$, $a < b$, $a = b$ or any other expression that has a truth associated with it.

See Also: `not, or`

Example

```
> 1<2 and 3<2;
```

$$false$$

```
> 1=1 and 3<4;
```

$$true$$

`arccos` Syntax: `arccos(`$expression$`)`

The inverse cosine function of $expression$. The answer is returned in radians.

`arccot` Syntax: `arccot(`*expression*`)`

The inverse cotangent function of *expression*. The answer is returned in radians.

`arccsc` Syntax: `arccsc(`*expression*`)`

The inverse cosecant function of *expression*. The answer is returned in radians.

`arcsec` Syntax: `arcsec(`*expression*`)`

The inverse secant function of *expression*. The answer is returned in radians.

`arcsin` Syntax: `arcsin(`*expression*`)`

The inverse sine function of *expression*. The answer is returned in radians.

`arctan` Syntax: `arctan(`*expression*`)` or `arctan(`$expression_1, expression_2$`)`

The first syntax above returns the arctangent of the *expresion*. The second syntax above returns the principal value of the argument of the complex number

$$expression_2 + expression_1 \cdot i,$$

which is such that $-\pi <$ `arctan(`$expression_1$`,`$expression_2$`)` $\leq \pi$. All answers are returned in radians.

Example

```
> arctan(1/(-1));
```

$$-\frac{1}{4}\pi$$

```
> arctan(1,-1);
```

$$\frac{3}{4}\pi$$

`ceil` Syntax: `ceil(`*expression*`)`

The `ceil` function returns the smallest integer that is greater than or equal to the value of the *expression*. It is related to the `floor` function by the relation `-floor(-x) = ceil(x)`.

See Also: `floor`

Example

```
> ceil(1.5);
```

$$2$$

changevar Syntax: `changevar(`*equation*, *expression*, *variable*`)`

Requires: `with(student)`

The `changevar` command allows the user to make a change of variables in the given *expression* where the given *equation* describes the relationship between the old variable and the new variable. The *variable* listed at the end of the command should be the variable that the user wants the new expression to be in terms of.

Note: `changevar` understands limits and sums as well.

Example Below, is an example wehre a change of variables is performed on an integral. Note that the equation used in the `changevar` command, $u = \ln(x)$, is just the kind of equation that one would write when performing a change of variables by hand.

```
> a := Int(ln(x)/x,x);
```

$$a := \int \frac{\ln(x)}{x} dx$$

```
> with(student):
> b := changevar(u=ln(x),a,u);
```

$$b := \int u \cdot du$$

```
> c := value(b);
```

$$c := \frac{1}{2}u^2$$

```
> d := subs(u=ln(x),c);
```

$$d := \frac{1}{2}\ln(x)^2$$

coeff Syntax: `coeff(`p, x, n`)`

Syntax: `coeff(`$p, x\hat{\ } n$`)`

The `coeff` command takes a polynomial p and picks out the coefficient of the term containing the variable x to the given power n.

WARNING: If like terms have not been collected together (i.e. there are perhaps two terms of x) then the user may need to use the command `collect` before using the `coeff` command.

WARNING: To return the constant term, the syntax `coeff(p,x,0)` must be used.

See Also: `collect`

Example

```
> f := (a+1)*x + (a+2)*x^2 + Pi;
```

$$f := (a+1)x + (a+2)x^2 + \pi$$

```
> coeff(f,x,2);
```

$$a+2$$

```
> coeff(f,x^2);
```

$$a+2$$

```
> coeff(f,x,0);
```

$$\pi$$

`collect` Syntax: `collect(`p`, `x`)`

The `collect` command is used to collect like terms of the variable x in the polynomial p.

WARNING: The variable x may not be raised to an exponent, i.e. `collect` works for x, but not x^2. A use of `collect` with the variable x will collect the x^2 terms as well as any other powers of x.

Example

```
> f := (a+1)*x + a^2*x + a;
```

$$f := (a+1)x + a^2x + a$$

```
> collect(f,x);
```

$$(a+1+a^2)x + a$$

```
> collect(f,a);
```

$$a^2x + (x+1)a + x$$

`combine` Syntax: `combine(`x`)` or `combine(`x`,` *option*`)`

The `combine` command allows the user to combine the terms of the expression x that **Maple V** doesn't combine on its own. The user may specify an extra *option* where *option* is one or more of the following: `exp, ln, power, trig, Psi, radical, abs, signum, plus, atatsign, conjugate, plot, product,` and `range`. If more than one item is chosen from this list, they must be enclosed in square brackets as a list. (See example below).

Example

```
> f := exp(sin(a)*cos(b))*exp(cos(a)*sin(b));
```

$$f := e^{(\sin(a)\cos(b))}e^{(\cos(a)\sin(b))}$$

```
> combine(f,exp);
```

$$e^{(\sin(a)\cos(b)+\cos(a)\sin(b))}$$

```
> combine(f,[trig,exp]);
```

$$e^{\sin(a+b)}$$

`completesquare` Syntax: `completesquare(`*expression*, x`)`

Requires: `with(student)`

The `completesquare` command attempts to complete the square on the given *expression* using the variable x.

Example

```
> with(student):
> f := x^2 + x + y^2 - 5*y + 9;
```

$$f := x^2 + x + y^2 - 5y + 9$$

```
> g := completesquare(f,x);
```

$$\left(x + \frac{1}{2}\right)^2 + \frac{35}{4} + y^2 - 5y$$

```
> h := completesquare(g,y);
```

$$\left(y - \frac{5}{2}\right)^2 + \frac{5}{2} + \left(x + \frac{1}{2}\right)^2$$

conjugate

Syntax: `conjugate(`x`)`

With the `conjugate` command, a user may ask for the complex conjugate of an expression x.

WARNING: All variables are assumed to be *complex* so `conjugate` returns variables unevaluated.

See Also: `evalc`

Example

```
> conjugate(1+I);
```

$$1 - I$$

```
> conjugate(exp(I));
```

$$e^{(-I)}$$

convert

Syntax: `convert(`x`,` *type*`)` or `convert(`x`,` *type*`,` *options*`)`

The `convert` command takes some expression x and tries to change it into the *type* specified. This is useful to a calculus student in these situations:

1. Converting sin, cos, tan, etc. into exponential forms.
2. Converting exponential forms into sin, cos, tan, etc.
3. Converting rational expressions (i.e. a polynomial with integer coefficients over another polynomial with integer coefficients) into partial fractions.

Example Convert trigonometric expression into exponential form.

```
> a := sin(x);
```

$$a := \sin(x)$$

```
> convert(a,exp);
```

$$-\frac{1}{2} I \left(e^{Ix} - \frac{1}{e^{(Ix)}} \right)$$

Example Convert exponential expression into trigonometric form.

```
> a := exp(I*x);
```

$$a := e^{(Ix)}$$

```
> convert(a,trig);
```

$$\cos(x) - I\sin(x)$$

$\boxed{\text{Example}}$ Convert a rational expression to partial fractions.

```
> a := -3/(x^2 + 5*x + 4);
```

$$a := -3\frac{1}{x^2 + 5x + 4}$$

```
> convert(a,parfrac,x);
```

$$\frac{1}{x+4} - \frac{1}{x+1}$$

Note the extra part of the partial fractions command. The final x in the command is part of the extra options for the `convert` command that allows you to specify which variable **Maple V** should use to decompose the expression into its parts.

For more help: Other options with the `convert` command can be found with the command:

```
> ?convert
```

`cos` Syntax: `cos(`x`)`

The `cos` function returns the cosine of the expression x where x is in radians.

`cot` Syntax: `cot(`x`)`

The `cot` function returns the cotangent of the expression x where x is in radians.

`csc` Syntax: `csc(`x`)`

The `csc` function returns the cosecant of the expression x where x is in radians.

`D` Syntax: `D(`f`)`

Syntax: `D[`v_1`](`f`)`

Syntax: `D[`$v_1, v_2, \ldots$`](`f`)`

The `D` is an operator that operates on f where f is a *function* and returns the derivative as a function. In the first syntax above, f must be a function of only one variable. The second syntax above is for a multivariate function, and v_1 is the number of the variable in line of the definition of the function. That is, if f is defined as:

```
> f := (x,y,z) -> sin(x)*sin(y)*sin(z);
```

$$f := (x, y, z) \to \sin(x)\sin(y)\sin(z)$$

Then we can differentiate with respect to z by letting $v_1 = 3$ since z is the third variable listed in the definition of f.

```
> fz := D[3](f);
```

$$fz := (x, y, z) \to \sin(x)\sin(y)\cos(z)$$

The third syntax above allows you to differentiate with respect to variables in the order listed. To differentiate with respect to x and then differentiate the result with respect to y in the f defined above:

```
> fxy := D[1,2](f);
```

$$fxy := (x, y, z) \to \cos(x)\cos(y)\sin(z)$$

The `D` operator is the analogue of the **`diff`** function, except that **`diff`** works on expressions and `D` operates on functions.

WARNING: If the function is defined by some other Maple commands other than a single assignment, then `D` may not work. Here is an example:

$\boxed{\text{Example}}$

```
> f := x -> exp(x)/(1+exp(x));
```

$$f := x \to \frac{e^x}{1+e^x}$$

```
> finv := x -> solve(f(y)=x,y);
```

$$finv := x \to solve(f(y) = x, y)$$

```
> D(finv);
```

$$D(finv)$$

In this example, `D` tries to differentiate a **`solve`** command, which doesn't make any sense. So **Maple V** returns the expression unevaluated.

__**See Also**__: **`diff`**

`denom` Syntax: `denom(`*expression*`)`

The `denom` command returns the **denom**inator of the given *expression*. If *expression* is not in normal form (see: `normal`) then **Maple V** puts *expression* in normal form and returns the denominator of the normal form.

See Also: `normal, numer`

Example

```
> f := x^2/(x^3-x-1);
```

$$f := \frac{x^2}{x^3 - x - 1}$$

```
> denom(f);
```

$$x^3 - x - 1$$

```
> g := sin(x)/cos(x);
```

$$g := \frac{\sin(x)}{\cos(x)}$$

```
> denom(g);
```

$$\cos(x)$$

`diff` Syntax: `diff(`*expression*`, `x`)`

Syntax: `diff(`*expression*`, `x_1`, `x_2`, `$\cdots$` , `x_n`)`

In the first form listed above, the `diff` command is used to **diff**erentiate the given *expression* with respect to the variable x.

WARNING: Note that `diff` will not differentiate a function.

In the second form listed above, `diff` is used to differentiate the *expression* with respect to the variables $x_1, x_2, \cdots, x_n$ in that order. Note that the variables $x_1, x_2, \cdots, x_n$ need not all be different and need not be in the expression (although this will likely result in a derivative of zero).

See Also: `D`

Example

```
> f := x^2 + x + 1;
```

$$f := x^2 + x + 1$$

```
> df := diff(f,x);
```

$$df := 2x + 1$$

Here is the way to find the second derivative of $\cos(x)$. Note the two x's.

```
> diff(cos(x),x,x);
```

$$-\cos(x)$$

`display` Syntax: `display(`$\{plot_1, plot_2, \cdots, plot_n\}$`)`

Requires: `with(plots)`

The `display` command can be used to combine several plots into one plot. The usual way to do this is to create the plots first and assign them to variables and then to "overlay" them with the `display` command.

Example This example creates three plots, assigns each to a variable, and then shows them together on one graph with the `display` command. Pay careful attention to the use of the colon at the end of the first four lines below. This suppresses the **Maple V** output. Simply put, it tells **Maple V** to perform the work, but not to show any results on the screen. In this way, the first four lines load the `plots` package and then create three plots and name them `a, b` and `c` without showing any results. The last command, however, does have a semicolon and so it shows the graph.

```
> with(plots):
> a := plot(sin(x),x=0..2*Pi):
> b := plot(cos(x),x=0..2*Pi):
> c := plot(sin(x-Pi/4),x=0..2*Pi):
> display({a,b,c});
```

`do` Syntax: `do` *statements* `od;`

The `do` command is used to enclose a sequence of *statements* that comprise the body of a loop defined with the `while` or the `for` commands.

See Also: `for, next, while`

`done` Syntax: `done`

The `done` command exits a **Maple V** session. It uses no parentheses and it does not require a semicolon or colon to be activated. Typing simply `done` and a carriage return will exit **Maple V** . It is equivalent to `quit` or `stop`.

`dsolve`

Syntax: `dsolve(`$deqn, x, options$`)`

Syntax: `dsolve(`$\{deqn, IC_1, IC_2, \cdots, IC_n\}, x, options$`)`

Syntax: `dsolve(`$\{deqn_1, deqn_2, \cdots, deqn_n\}$`,`$\{x_1, x_2, \cdots, x_n\}, options$`)`

Syntax: `dsolve(`$\{deqn_1, \cdots, deqn_n, IC_1, \cdots, IC_m\}, \{x_1, \cdots, x_n\}, options$`)`

Maple V solves differential equations with `dsolve`. Each of the above is a legal way to call the `dsolve` command. The equation(s) can be specified, as well as any initial condition(s). The variable(s) that are to be solved for are given second in the command and any other options are listed after that. **Maple V** returns an equation as an answer. To extract the right hand side of the equation, use the `rhs` command.

Two common options are `laplace` and `numeric`. The `laplace` option forces **Maple V** to attempt to solve the differential equation(s) with the Laplace method. The `numeric` option forces **Maple V** to return a procedure that can be used to find numeric solutions to the given equation(s). If numeric solutions are desired, then initial conditions are *not* optional; they must be specified. Also, when using the `numeric` option, the answer is returned as a procedure that returns a list containing values of the variable(s) at the specified point and values of derivatives at that point.

See Also: `D, diff, rhs`

Example In this example, the differential equation

$$y'' - y = e^x$$

is solved with the initial conditions

$$y'(0) = -1, y(0) = 1$$

.

```
> eqn1 := (D@@2)(y)(x) - y(x) = exp(x);
```

$$eqn1 := D^{(2)}(y)(x) - y(x) = e^x$$

```
> IC := D(y)(0) = -1, y(0)=1;
```

$$D(y)(0) = -1, y(0) = 1$$

```
> sol1 := dsolve(eqn1, y(x));
```

$$sol1 := y(x) = \frac{1}{2}xe^x - \frac{1}{4}e^x + \_C1e^x + \_C2e^{(-x)}$$

```
> sol2 := dsolve({eqn1, IC}, y(x));
```

$$sol2 := y(x) = \frac{1}{2}xe^x - \frac{1}{4}e^x + \frac{5}{4}e^{(-x)}$$

```
> sol3 := dsolve({eqn1, IC}, y(x), laplace)
```

$$sol3 := y(x) = \frac{1}{2}xe^x - \frac{1}{4}e^x + \frac{5}{4}e^{(-x)}$$

```
> sol4 := dsolve({eqn1, IC}, y(x), numeric);
sol4 := proc(rkf45_x) ...  end
```

```
> sol4(0);
```

$$[x = 0, y(x) = 1., \frac{\partial}{\partial x}y(x) = -1.]$$

```
> sol4(1);
```

$$[x = 1, y(x) = 1.139419423335927, \frac{\partial}{\partial x}y(x) = 1.578861289040924]$$

`eval` Syntax: `eval(`*expression*`)`

The `eval` command forces **eval**uation of unevaluated subexpressions within *expression*. This command may be needed after a `subs` command, since `subs` does not force evaluation after a substitution.

See Also: `evalb, evalc, evalf, value`

Example

```
> f := sin(x);
```

$$f := \sin(x)$$

```
> f2 := subs(x=Pi,f);
```

$$f2 := \sin(\pi)$$

```
> eval(f2);
```

$$0$$

`evalb` Syntax: `evalb(`*expression*`)`

The `evalb` command forces evaluation of an expression that has some truth value associated with it. In fact, the name `evalb` is short for **eval**uate **b**oolean, which implies that it should evaluate boolean expressions.

See Also: `and, eval, not, or`

Example

```
> 10<15;
```

$$10 < 15$$

```
> evalb(10<15);
```

$$true$$

```
> evalb(10=15);
```

$$false$$

`evalc` Syntax: `evalc(`*expression*`)`

The `evalc` command forces evaluation of *expression* and assumes that any variables in the *expression* are real-valued, i.e. they don't contain any complex numbers. Also, it returns the expression in the canonical form $a + bI$ whenever possible.

See Also: `eval`

Example Consider the output from the following.

```
> f := exp(a+b*I);
```

$$f := e^{(a+bI)}$$

```
> evalc(f);
```

$$e^a \cos(b) + I e^a \sin(b)$$

`evalf` Syntax: `evalf(`*expression*`)`

Syntax: `evalf(`*expression*`,` n`)`

`evalf` is shorthand for **eval**uate as a **f**loating point number. It is used to force **Maple V** to find a decimal approximation of the *expression* that is given.

`evalf` can be called in either of the two ways listed above. The first way asks for a decimal approximation of *expression* using the current value of `Digits` (see `Digits`). The second way asks for a decimal approximation of *expression* with n digits of accuracy.

<u>**See Also**</u>**:** `Digits, eval`

Example

```
> f := sin(1);
```

$$f := \sin(1)$$

```
> evalf(f);
```

$$.8414709848$$

```
> evalf(f,20);
```

$$.84147098480789650665$$

`example` Syntax: `example(`*command*`)`

With the `example` command the user can call up an example of the given *command* from the help page. This is equivalent to `??`*command*.

`exp` Syntax: `exp(`*expression*`)`

The `exp` function is the **exp**onential function.

<u>**See Also**</u>**:** `E` in chapter 3

`expand` Syntax: `expand(`*expression*`)`

Syntax: `expand(`$expression, subexpression_1, \cdots, subexpression_n$`)`

The `expand` command expands an *expression* according to known mathematical conventions. It knows how to expand most built-in functions like **cos, sin, tan, sinh, cosh, tanh, exp, ln, sum, product, int, limit,** and **abs**.

The second syntax above allows the user to specify what subexpressions in *expression* not to expand.

Example

```
> f := sin(x+y);
```

$$f := \sin(x + y)$$

```
> expand(f);
```

$$\sin(x)\cos(y) + \cos(x)\sin(y)$$

```
> g := exp(x+y);
```

$$g := e^{(x+y)}$$

```
> expand(g);
```

$$e^x e^y$$

```
> h := (x+y)*(x+5*y);
```

$$h := (x + y)(x + 5y)$$

```
> expand(h);
```

$$x^2 + 6xy + 5y^2$$

```
> expand(h,x+5*y);
```

$$(x + 5y)x + (x + 5y)y$$

factor Syntax: `factor(`*expression*`)`

The `factor` command factors the given *expression*. If the *expression* does not factor, then the *expression* is returned unevaluated. This does not factor integers into primes.

Example

```
> f := x^5 + 14*x^4 + 38*x^3 -164*x^2 -519*x + 630;
```

$$f := x^5 + 14x^4 + 38x^3 - 164x^2 - 519x + 630$$

```
> factor(f);
```

$$(x + 5)(x + 6)(x + 7)(x - 1)(x - 3)$$

```
> factor(sin(x)^2 -1);
```

$$(\sin(x) - 1)(\sin(x) + 1)$$

factorial Syntax: `factorial(`*expression*`)`

`factorial` returns the factorial function of the given expression. If *expression* evaluates to a non-negative integer, then the value is computed. If it does not evaluate completely, then the function is returned in unevaluated form. The `factorial` function is equivalent to the `!` function.

`floor` Syntax: `floor(`*expression*`)`

The `floor` function returns the greatest integer that is less than or equal to the given *expression*.

See Also: `ceil`

Example

```
> floor(1.5);
```

$$1$$

```
> floor(-2.3);
```

$$-3$$

`for` Syntax: `for` x `from` $expression_1$ `to` $expression_2$ `by` $expression_3$ `do` *statements* `od`

Syntax: `for` x `in` *expression* `do` *statements* `od`

Syntax: `for` x `from` $expression_1$ `by` $expression_3$ `while` *condition* `do` *statements* `od`

The `for` command is used to begin a loop in **Maple V** . Below are descriptions of each form listed above:

The first syntax is the standard loop. It starts some variable x at the value of $expression_1$ and executes the *statements* listed between the `do` and `od`. Then the value of x is incremented by the amount $expression_3$ and if the new value of x is not more than $expression_2$ then the *statements* are executed again. This process continues until x has incremented through values between $expression_1$ and $expression_2$. When the loop finishes, it drops to the next executable statement below the `od`. The `from, to`, and `by` may be omitted and **Maple V** assumes their default values to be 1, infinity, and 1 respectively.

The second syntax above executes a loop that for each iteration lets x assume the value of each the operands of *expression*.

The third syntax above constructs a loop that steps x from $expression_1$ by steps of size $expression_3$ as long as *condition* evaluates as **true**. In this loop, as in the loops above, the `by` may be omitted and the step value is assumed to be 1.

See Also: `break, do, next, while`

Example This example prints the cubes of the numbers 1 to 4.

```
> for i from 1 to 4 do i^3 od;
```

$$1$$

$$8$$

$$27$$

$$64$$

$\boxed{\text{Example}}$ This example differentiates the elements of the set S.

```
> S := {x,x^2, x^3};
```

$$S := \{x, x^2, x^3\}$$

```
> for i in S do diff(i,x) od;
```

$$1$$

$$2x$$

$$3x^2$$

$\boxed{\text{Example}}$ This example adds the integers starting with 1 while the sum is less than 6.

```
> s := 0;
```

$$s := 0$$

```
> for i from 1 while s<6
> do
> s := s+i
> od;
```

$$s := 1$$

$$s := 3$$

$$s := 6$$

`fsolve` Syntax: `fsolve(`*equation*`,` x`)`

Syntax: `fsolve(`*equation*`,` $x = a..b$`)`

Syntax: `fsolve(`$\{equation_1, \cdots, equation_n\}$`,` $\{x_1, \cdots, x_n\}$`)`

The name of the `fsolve` command is an abbreviation for **f**loating point number **solve**. This means, in plain terms, find a decimal approximation of a solution to the equation(s) that the user supplies. One very important thing to know about the `fsolve` command is that it must solve for as many variables as there are variables in the equations.

The first syntax above finds a numeric solution for the variable x in the given *equation*.

The second syntax above finds a numeric solution for the variable x in the given *equation* and forces **Maple V** to search for solutions only in the range $[a, b]$.

The third syntax above finds numeric solutions for the variables $x_1, x_2, \cdots, x_n$ given the equations $equation_1, equation_2, \cdots, equation_n$.

See Also: `solve`

Example This example tries to find the first three solutions to $\sin(x^2) = 0$ for $x \geq 0$.

```
> eqn := sin(x^2)=0;
```

$$eqn := \sin(x^2) = 0$$

```
> fsolve(eqn, x);
```

$$2.506628275$$

This is really one of multiple solutions; **Maple V** found it on its own. The following finds another two solutions.

```
> fsolve(eqn,x=0..1);
```

$$0$$

```
> fsolve(eqn,x=1..2);
```

$$1.772453851$$

Example Here is an example with more than one variable. **Maple V** is finding a solution for x, y and z in the equations $\sin(xy) + \cos(xz) = 0, x + y + z = 1, x - y = 5$.

```
> eqn1 := sin(x*y) + cos(x*z)=0;
```

$$eqn1 := \sin(xy) + \cos(xz) = 0$$

```
> eqn2 := x+y+z=1;
```

$$eqn2 := x + y + z = 1$$

```
> eqn3 := x-y=5;
```

$$eqn3 := x - y = 5$$

```
> fsolve({eqn1,eqn2,eqn3},{x,y,z});
```

$$\{z = -9.008455550, x = 7.504227775, y = 2.504227775\}$$

Heaviside Syntax: `Heaviside(`*expression*`)`

The **Heaviside** function is a special function defined in the following way:

$$Heaviside(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

The **Heaviside** function can be used to create piecewise-defined functions.

Helpful Hint: The `alias` command is often used in the following way

```
> alias(H=Heaviside);
```

$$I, H$$

to allow the user to refer to the **Heaviside** function with just the shorthand **H** instead of **Heaviside**.

See Also: `alias`

Example

```
> alias(H=Heaviside);
```

$$I, H$$

```
> H(1);
```

$$1$$

```
> f := x -> H(x) - H(x-1);
```

$$f := x \to H(x) - H(x-1)$$

```
> f(1/2);
```

$$0$$

help Syntax: `help(`*command*`)`

This command is equivalent to `?`*command*.

`if...fi` Syntax: `if` *expression* `then` *statements* `fi`

Syntax: `if` *expression* `then` $statements_1$ `else` $statements_2$ `fi`

Syntax: `if` $expression_1$ `then` *statements* `elif` $expression_2$ `then ... fi`

The `if` command is used to set up conditional statements in **Maple V** . This means that the user wants a particular thing to happen if an *expression* is true and wants another thing to happen if the *expression* is false.

WARNING: In **Maple V** the `if` command must be followed at some point by its counterpart `fi` (i.e. "`if`" backwards).

The first syntax above is a simple conditional statement with the `if` command. **Maple V** first checks to see whether *expression* evaluates to true or false. If it is true, then the *statements* are executed. If it is false, the *statements* are not executed.

The second syntax above includes the `else` option to an `if` command. In this case, if *expression* evaluates to true, then $statements_1$ is executed and $statements_2$ is ignored. Otherwise, if *expression* if false, then **Maple V** ignores $statements_1$ and executes $statements_2$.

The third syntax includes another auxiliary command `elif` which is short for *else if*. In this case, if $expression_1$ is true, then *statements* is executed. If $expression_1$ is false, then **Maple V** ignores *statements* and tests $expression_2$ to see if it is true or false. More statements can follow the `elif` part and will be treated in the same way that they would be treated with a regular `if` command.

$\boxed{\text{Example}}$ In this example, the function

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ x^2 & \text{if } x < 0 \end{cases}$$

is defined with an `if` statement.

```
> f := x -> if x>=0 then x else x^2 fi;
f := proc(x) options operator,arrow; if 0 <= x then x else x^2 fi end
> f(2);
                                    2
> f(-2);
                                    4
```

info Syntax: `info(`*command*`)`

This command retrieves a short description of a *command*. The description given is really a part of the help page for the given *command*.

WARNING: Some commands must be enclosed in back quotes to be used with the `info` command.

Example To use `info` on the `done` command, the `done` command must be in back quotes as below:

```
> info(`done`);
```

int,Int Syntax: `int(`*expression*, x`)` indefinite integral

Syntax: `int(`*expression*, $x = a..b$`)` definite integral

The `int` command is the **Maple V int**egration command. **Maple V** recognizes a wide variety of integrals. It can be used to get exact values, or decimal approximations when an exact value cannot be computed.

The `Int` command follows the same syntax as the `int` command but **Maple V** doesn't evaluate the integral, it merely displays it. The `value` command can be used to get the value of an unevaluated integral.

WARNING: **Maple V** does not include an arbitrary constant of integration when it integrates.

Helpful Hint: Use the `evalf` command on an unevaluated definite integral to get a decimal approximation.

Example This example shows two different ways to perform the same indefinite integration. Here is the first.

```
> f := sin(x)+cos(x);
```

$$f := \sin(x) + \cos(x)$$

```
> g := Int(f,x);
```

$$\int \sin(x) + \cos(x) dx$$

```
> h := value(g);
```

$$h := -\cos(x) + \sin(x)$$

Here is a second way, involving just the `int` command.

```
> int(f,x);
```

$$-\cos(x)+\sin(x)$$

$\boxed{\text{Example}}$ This example shows two different ways to perform the same definite integration.

```
> f := x*exp(x);
```

$$f := xe^x$$

```
> g := Int(f,x=0..1);
```

$$\int_0^1 xe^x dx$$

```
> h := value(g);
```

$$h := 1$$

```
> int(f,x=0..1);
```

$$1$$

$\boxed{\text{Example}}$ This example shows how `evalf` must be used to get a approximation. When **Maple V** returns the `int` command unevaluated, it is an indication that you may want to use `evalf`.

```
> f := sin(t^3);
```

$$f := \sin(x^3)$$

```
> g := int(f,x=0..Pi);
```

$$\int_0^\pi \sin(x^3)dx$$

```
> h := evalf(g);
```

$$h := .4158338147$$

`integrand` Syntax: `integrand(`*expression*`)`

Requires: `with(student)`

The `integrand` command, when applied to an *expression* that contains an integral, returns the integrand of that integral. If it is applied to an equation with more than one integral, it returns a set containing the integrands of all the integrals.

$\boxed{\text{Example}}$

```
> with(student):
> a := Int((x+1)/x,x);
```

$$a := \int \frac{x+1}{x} dx$$

```
> b := integrand(a);
```

$$\frac{x+1}{x}$$

`intparts` Syntax: `intparts(`*expression*, u`)`

Requires: `with(student)`

The `intparts` command allows the user to perform **int**egration by **parts** on the *expression* given (provided that it contains an integral) by specifying the u part of the integration by parts.

Usually, with an integration by parts, you let a particular part of the integral be called u and then you find du, v, and dv based on that choice of u. **Maple V** can do the same operations on an integral if you decide which part to define as u.

Consider the equation

$$\int u \cdot dv = uv - \int v \cdot du$$

The part that you give as *expression* is the left hand side of the equation, and if you also specify what the u is in `intparts` command, **Maple V** should give you the right hand side of this equation.

<u>**See Also**</u>**:** `value`

$\boxed{\text{Example}}$ Here we calculate $\int x^2 e^x dx$.

```
> with(student):
> a := Int(x^2*exp(x),x);
```

$$a := \int x^2 e^x dx$$

Here we have let $u = x^2$ and **Maple V** performs the integration.

```
> b := intparts(a,x^2);
```

$$b := x^2 e^x - \int 2xe^x dx$$

Again we perform an integration by parts, but this time we let $u = 2x$.

```
> c := intparts(b,2*x);
```

$$c := x^2 e^x - 2xe^x + \int 2e^x \, dx$$

```
> d := value(c);
```

$$d := x^2 e^x - 2xe^x + 2e^x$$

leftbox Syntax: `leftbox(`*expression*, $x = a..b$`)`

Syntax: `leftbox(`*expression*, $x = a..b, n$`)`

Requires: `with(student)`

The **leftbox** command produces a plot of the given *expression* and the boxes used to compute a left Riemann sum. The first syntax above assumes that the user wants 4 boxes by default. The second syntax allows the user to specify the number of boxes with the integer n.

See Also: leftsum, middlebox, middlesum, rightbox, rightsum, student

leftsum Syntax: `leftsum(`*expression*, $x = a..b$`)`

Syntax: `leftsum(`*expression*, $x = a..b, n$`)`

Requires: `with(student)`

The **leftsum** command computes a left Riemann sum of the given *expression* in terms of the variable x between the values $x = a$ and $x = b$. These sums can approximate well the integral of the given *expression* if the number of intervals is sufficiently large. If the first syntax above is used, then the default number of intervals is 4. In the second syntax, the number n specifies the number of intervals to be used.

The answer is returned as a summation and may be evaluated with either **evalf**, or **value**.

See Also: `evalf, leftbox, middlebox, middlesum, rightbox, rightsum, student, value`

lhs Syntax: `lhs(`*equation*`)`

The **lhs** command is an abbreviation for **l**eft **h**and **s**ide. It returns the left hand side of the *equation* that the user supplies. **lhs** works for inequalities as well.

See Also: `rhs`

Example

```
> f := t+1 = sin(x) - exp(x);
```

$$f := t + 1 = \sin(x) - e^x$$

```
> lhs(f);
```

$$t + 1$$

limit, Limit Syntax: `limit(`$expression, x = a$`)`

Syntax: `limit(`$expression, x = a, direction$`)`

The `limit` command is used to find the limit of the given *expression* as the variable x approaches the value a. If a limit from a particular direction is desired, the user should supply the third option *direction* where *direction* can be `left` or `right`.

The `Limit` command uses the same syntax as the `limit` command but returns an unevaluated limit. The unevaluated limit can be evaluated using the `value` command.

See Also: `value`

Example Take the limit of $\frac{\sin(x)}{x}$ as x approaches zero.

```
> f := sin(x)/x;
```

$$f := \frac{\sin(x)}{x}$$

```
> g := Limit(f,x=0);
```

$$g := \lim_{x \to 0} \frac{\sin(x)}{x}$$

```
> value(g);
```

$$1$$

```
> limit(f,x=0);
```

$$1$$

```
> limit(exp(-x),x=infinity);
```

$$0$$

Example Take the limit of `Heaviside(`x`)` as x approaches 0 from the right.

```
> f := Heaviside(x);
```

$$f := Heaviside(x)$$

```
> limit(f,x=0);
```

$$undefined$$

```
> limit(f,x=0,right);
```

$$1$$

`ln` Syntax: `ln`(*expression*)

The `ln` function is the natural logarithm. It is the logarithm with base e.

See Also: `log`

`log` Syntax: `log`(*expression*)

Syntax: `log[`b`]`(*expression*)

The `log` function is equivalent to the function `ln` if there is no b specified in square brackets (this is the first syntax above). When the second syntax above is used, **Maple V** returns the logarithm of the *expression* in the base b.

See Also: `ln`

Example The log of 10 in base e and base 10.

```
> evalf(log(10));
```

$$2.302585093$$

```
> evalf(log[10](10));
```

$$1.$$

`map` Syntax: `map`(f, *expression*)

The `map` command is used to apply some function to the parts of an expression. The function f can be a function from **Maple V** or one that the user has defined. The `map` command is best understood by examples.

Example In this example, the `abs` (absolute value) function is applied to the elements of a list.

```
> LIST := [1,-2,3,-4];
```

$$LIST := [1, -2, 3, -4]$$

```
> map(abs,LIST);
```

$$[1, 2, 3, 4]$$

$\boxed{\text{Example}}$ In this example, a function is defined by the user and then is applied to the elements of the list in the above example.

```
> myfunc := x -> x^2+5;
```

$$myfunc := x \to x^2 + 5$$

```
> map(myfunc, LIST);
```

$$[6, 9, 14, 21]$$

$\boxed{\text{Example}}$ In this example, the function is defined right in the `map` command itself.

```
> map(x->x^2+5,LIST);
```

$$[6, 9, 14, 21]$$

`middlebox`

Syntax: `middlebox(`$expression, x = a..b$`)`

Syntax: `middlebox(`$expression, x = a..b, n$`)`

Requires: `with(student)`

The `middlebox` command produces a plot of the given *expression* and the boxes used to compute a middle Riemann sum. The first syntax above assumes that the user wants 4 boxes by default. The second syntax allows the user to specify the number of boxes with the integer n.

$\boxed{\text{Example}}$ In this example, the middle sum of the function

$$f(x) = 1 - x - x^2$$

is represented between $x = 0$ and $x = 1/2$. First the default four intervals are used and then 15 intervals are used. Note that the function $f(x)$ is entered as the expression `f`, since `middlebox` expects an expression.

See Also: `leftbox, leftsum, middlesum, rightbox, rightsum, student`

```
> with(student):
> f := 1-x-x^2;
```

$$f := 1 - x - x^2$$

```
> middlebox(f,x=0..1/2);
> middlebox(f,x=0..1/2,15);
```

middlesum

Syntax: middlesum($expression$, $x = a..b$)

Syntax: middlesum($expression$, $x = a..b, n$)

Requires: with(student)

The middlesum command computes a middle Riemann sum of the given *expression* in terms of the variable x between the values $x = a$ and $x = b$. These sums can approximate well the integral of the given *expression* if the number of intervals is sufficiently large. If the first syntax above is used, then the default number of intervals is 4. In the second syntax, the number n specifies the number of intervals to be used.

The answer is returned as a summation and may be evaluated with either evalf, or value.

See Also: leftbox, leftsum, middlebox, rightsum, rightbox, student, value

Example In this example, the middle sum of the function

$$f(x) = 1 - x - x^2$$

is found between $x = 0$ and $x = 1/2$. First the default four intervals are used and then 15 intervals are used. Note that the function $f(x)$ is entered as the expression f, since middlesum expects an expression.

```
> with(student):
> f := 1-x-x^2;
```

$$f := 1 - x - x^2$$

```
> s1 := middlesum(f,x=0..1/2);
```

$$s1 := \frac{1}{8}\left(\sum_{i=0}^{3}\left(\frac{15}{16} - \frac{1}{8}i - \left(\frac{1}{8}i + \frac{1}{16}\right)^2\right)\right)$$

```
> value(s1);
```

$$\frac{171}{512}$$

```
> evalf(s1);
```

$$.3339843750$$

```
> s2 := middlesum(f,x=0..1/2,15);
```

$$s2 := \frac{1}{30}\left(\sum_{i=0}^{14}\left(\frac{59}{60} - \frac{1}{30}i - \left(\frac{1}{30}i + \frac{1}{60}\right)^2\right)\right)$$

```
> value(s2);
```

$$\frac{7201}{21600}$$

```
> evalf(s2);
```

$$.3333796296$$

next Syntax: `next`

The `next` command forces a `do` loop to continue to the next iteration. Within nested loops, it will proceed with the innermost nested loop. The `next` command *must* be within a loop; if it is not within a loop, an error will occur.

See Also: `do, for, while`

nops Syntax: `nops(`*expression*`)`

The `nops` command returns the **n**umber of **op**erands in the given *expression*. If the *expression* is a sum of terms, i.e. it has the + sign in it, then the `nops` command returns the number of terms being added. The `nops` command works similarly for products, exponentiations, lists, arrays, and other data types. The `nops` command is especially useful for writing procedures.

See Also: `op`

Example

```
> f := a + b + c;
```

$$f := a + b + c$$

```
> nops(f);
```

$$3$$

```
> g := a*b*c*d;
```

$$g := abcd$$

```
> nops(g);
```

$$4$$

```
> h := a^b;
```

$$h := a^b$$

```
> nops(h);
```

$$2$$

normal Syntax: `normal(`*expression*`)`

For a given *expression*, the `normal` command returns the "normal" form, i.e. numerator over denominator. Usually this is used with polynomials and rational expressions, however, it works for general expressions as well.

Example

```
> f := 1/x + x^2 - 1;
```

$$f := \frac{1}{x} + x^2 - 1$$

```
> normal(f);
```

$$\frac{1 + x^3 - x}{x}$$

```
> g := sin(x) + 1/sin(x)^2;
```

$$g := \sin(x) + \frac{1}{\sin(x)^2}$$

```
> normal(g);
```

$$\frac{\sin(x)^3 - 1}{\sin(x)^2}$$

not Syntax: `not` *expression*

The `not` command is used to negate the truth value of a given expression. This is best seen by example.

See Also: `and, evalb, or`

Example In this example, the value of the variable x is set to 10 and then various tests are performed using the `not` command.

```
> x := 10;
```

$$x := 10$$

```
> evalb(x < 15);
```

$$true$$

```
> evalb(not x < 15);
```

$$false$$

```
> evalb(x = 11);
```

$$false$$

```
> evalb(not x = 11);
```

$$true$$

`numer` Syntax: `numer(`*expression*`)`

The `numer` command returns the numerator of the given *expression*. If the *expression* is not in the form numerator over denominator, **Maple V** applies the `normal` command to the *expression* and then returns the denominator of that expression.

See Also: `denom, normal`

Example

```
> f := x*y/(1+x);
```

$$f := \frac{xy}{1+x}$$

```
> numer(f);
```

$$xy$$

```
> g := 1/x + 1;
```

$$g := \frac{1}{x} + 1$$

```
> numer(g);
```

$$1 + x$$

```
> normal(g);
```

$$\frac{1+x}{x}$$

`or` Syntax: $expression_1$ `or` $expression_2$

The `or` is a logical connector between two conditions $expression_1$ and $expression_2$ where $expression_1$ and $expression_2$ may be conditions like $a > b$, $a < b$, and $a = b$ or any other expression that has a truth associated with it.

See Also: `and, evalb, not`

`op` Syntax: `op(`*expression*`)`

Syntax: `op(`$n, expression$`)`

The `op` command allows the user to pick apart expressions into fragments called operands.

The first syntax listed above asks **Maple V** to respond with an expression sequence of all of the operands of the given *expression*.

The second syntax returns the operand in the n^{th} place in the sequence from `op(`$expression$`)`. That is, it returns the n^{th} operand.

See Also: `nops`

`order` Syntax: `order(`$series$`)`

The order command returns the order of a given *series*. Note that the *series* here is a special type of series in **Maple V** which involves a term $O(x^n)$ where the n is the order term of a series which represents the indeterminate part of the series.

Example Here **Maple V** first computes a series for $\sin(x)$ and then returns the order with the `order` command.

```
> s := series(sin(x),x);
```

$$s := x - \frac{1}{6}x^3 + \frac{1}{120}x^5 + O(x^6)$$

```
> order(s);
```

$$6$$

`piecewise` Syntax: `piecewise(`$m, relation_1, expression_1, \cdots, relation_n, expression_n$`)`

Syntax: `piecewise(`$m, relation_1, expression_1, \cdots, relation_n, expression_n, expression_{n+1}$`)`

Requires: `readlib(piecewise)`

The `piecewise` function, once it has been read into **Maple V** with the `readlib` command, can be used to easily define piecewise functions. When a function defined with `piecewise` is evaluated at a given point, **Maple V** looks at all of the relations, $relation_1, relation_2, \cdots, relation_n$, to find which one applies to the value that was given, i.e. which relation evaluates to true. The *expression* following the true relation is then used to determine the value of the function.

If the second syntax above is used, then $expression_{n+1}$ determines the value of the function at any value of the variable not covered in any of the n relations.

The number m can be used to define the smoothness of the function at the points between different pieces. If m is -1 then it is understood that the function is not continuous at the joints.

Example In this example, a function $f(x)$ is defined where $f(x) = sin(x)$ when $x \geq 0$ and $f(x) = x$ when $x < 0$.

```
> readlib(piecewise);
```

`proc(n,a,t,f) ... end`

```
> f := x -> piecewise(1, x < 0, x, x >= 0, sin(x));
```

$$f := x \to piecewise(1, x < 0, x, 0 \leq x, sin(x))$$

```
> f(-1);
```

$$-1$$

```
> f(0);
```

$$0$$

```
> f(Pi/2);
```

$$1$$

plot Syntax: `plot(`$f, a..b, c..d$`)`

Syntax: `plot(`$expression, x = a..b, y = c..d$`)`

The `plot` command is the versatile **Maple V** command that allows the user to create two-dimensional graphs of suitable functions or expressions. There are many ways to use the `plot` command, but the two main ways are outlined here.

The first syntax for `plot` that is listed above is the one to use when plotting a function. The function f is listed along with the range $a..b$ of values for the variable of f. You should not specify a variable for the range, but rather just the values that it will range between. The range $a..b$ specifies the values along the normal x-axis. Optionally, the user may include a vertical range $c..d$ which specifies the values along the normal y-axis. If the user does not specify the second range, **Maple V** will try to provide a suitable one from the values that the function f will take between a and b.

See Also: display, plots

Example Here are three ways to plot the sine function as a function.

```
> plot(sin, -Pi..Pi, -2..2);

> plot(sin, -Pi..Pi);

> plot(sin);
```

The last command has an understood domain of -10 to 10, and **Maple V** tries to find suitable values for the range (i.e. the y-axis).

The second syntax for `plot` that we cover here is for plotting expressions. The syntax is basically the same, except that the user must specify the variable in the *expression*. As with functions, the user need not specify a range along the vertical axis unless so desired.

$\boxed{\text{Example}}$ Here we plot the sine function as an expression in two different ways. In the second example, **Maple V** will choose the values for the vertical axis.

```
> plot(sin(x), x=-Pi..Pi, y=-2..2);
```

```
> plot(sin(x), x=-Pi..Pi);
```

In either of the above types, a user may plot more than one function or expression on the same graph with the use of the curly braces { and }.

$\boxed{\text{Example}}$ Here we plot sin(x), cos(x), and tan(x) all on the same graph. First we do it with functions, then with expressions.

```
> plot({sin, cos, tan}, -2*Pi..2*Pi, -5..5);
```

```
> plot({sin(x), cos(x), tan(x)}, x=-2*Pi..2*Pi, y=-5..5);
```

When plotting functions or procedures that have conditional statements (i.e. `if ... then ... fi` statements) in them, it may be necessary to keep the function or procedure in unevaluated form. This is attained by using the ' mark.

$\boxed{\text{Example}}$ The function $f(x)$ is $\sin(x)$ for $x > 0$ and x^2 for $x \leq 0$. The first attempt generates an error, but the second attempt is a correct way to plot the function.

```
> f := x -> if x<=0 then x^2 else sin(x) fi;
```

```
f := proc(x) options operator,arrow; if x <= 0 then x^2 else sin(x) fi end
```

```
> plot(f(x), x=-2..2);
Error, (in f) cannot evaluate boolean
```

```
> plot('f(x)', x=-2..2);
```

This problem can be avoided altogether with the command:

```
> plot(f, -2..2);
```

There are various options available with `plot`. A list of them can be found by typing

```
> ?plot[options]
```

plots Syntax: `with(plots)`

The `plots` package contains many commands useful for graphing functions in one or two dimensions. A full detail of its entries can be obtained with `> ?plots`.

<u>**See Also**</u>**:** `display, with`

proc Syntax: `proc(`$x_1, x_2, \cdots, x_n$`)` *statements* `end`

The `proc` command is used to produce a **proc**edure that accepts $x_1, x_2, \cdots, x_n$ arguments and then executes a body of *statements* until it reaches the `end` command.

The `proc` command is an alternate way to create a function, and is the main programming structure in **Maple V** . Its usefulness and variety are too extensive for a complete discussion here.

$\boxed{\text{Example}}$ Here is a procedure that reads in a list of 5 numbers and returns a polynomial with those numbers as roots.

```
> poly := proc(a,b,c,d,e)
> local ppoly;
> ppoly := expand((x-a)*(x-b)*(x-c)*(x-d)*(x-e));
> RETURN(ppoly);
> end;
> poly(1,2,3,4,4);
```

$$x^5 - 14x^4 + 75x^3 - 190x^2 + 224x - 96$$

product Syntax: `product(`$f(i), i = m..n$`)`

The `product` command produces a product of terms based on the function f and the range $i = m..n$. The resulting product is $f(m) \cdot f(m+1) \cdot f(m+2) \cdots f(n-1) \cdot f(n)$. The syntax listed is the syntax used to create

$$\prod_{i=m}^{n} f(i).$$

To create an unevaluated product, use the `Product` command in place of the `product` command. Use the value command to evaluate a `Product`.

<u>**See Also**</u>**:** `value`

$\boxed{\text{Example}}$ This example finds the product of factors of the form $(x-i)$ where i ranges from 2 to 10.

```
> Product((x-i),i=2..10);
```

$$\prod_{i=2}^{10}(x-i)$$

```
> product((x-i),i=2..10);
```

$$(x-2)(x-3)(x-4)(x-5)(x-6)(x-7)(x-8)(x-9)(x-10)$$

`quit` Syntax: `quit`

This command quits the **Maple V** system. It does not require a semicolon or a colon at the end. It is equivalent to `done` or `stop`.

`readlib` Syntax: `readlib(`*command*`)`

Some functions which are not loaded automatically with **Maple V** can be loaded with the `readlib` command. An example of this is the `piecewise` command.

See Also: `piecewise`

`related` Syntax: `related(`*command*`)`

The `related` command returns a list of related topics for the *command* specified. The returned information is the information found at the bottom of the help page for *command* under `SEE ALSO`.

`restart` Syntax: `restart`

This command restarts the **Maple V** system. Using the `restart` command will clear all assigned variables and procedures and detach any packages previously known to **Maple V** . It is not the case that all memory will be returned as free. For maximum memory, it is best to close the current **Maple V** session and start with a fresh one.

`rightbox` Syntax: `rightbox(`$expression, x = a..b$`)`

Syntax: `rightbox(`$expression, x = a..b, n$`)`

Requires: `with(student)`

The `rightbox` command produces a plot of the given *expression* and the boxes used to compute a right Riemann sum. The first syntax above assumes that the user wants 4 boxes by default. The second syntax allows the user to specify the number of boxes with the integer n.

See Also: `leftbox, leftsum, middlebox, middlesum, rightsum, student`

`rightsum`

Syntax: `rightsum(`$expression, x = a..b$`)`

Syntax: `rightsum(`$expression, x = a..b, n$`)`

Requires: `with(student)`

The `rightsum` command computes a right Riemann sum of the given *expression* in terms of the variable x between the values $x = a$ and $x = b$. These sums can approximate well the integral of the given *expression* if the number of intervals is sufficiently large. If the first syntax above is used, then the default number of intervals is 4. In the second syntax, the number n specifies the number of intervals to be used.

The answer is returned as a summation and may be evaluated with either `eval`, `evalf`, or `value`.

See Also: `leftbox, leftsum, middlebox, middlesum, rightbox, student, value`

`rhs`

Syntax: `rhs(`*equation*`)`

The `rhs` command will return the **r**ight **h**and **s**ide of the given *equation*.

See Also: `lhs`

Example

```
> eqn := y(x) = cos(x) + 1/x;
```

$$eqn := y(x) = \cos(x) + \frac{1}{x}$$

```
> rhs(eqn);
```

$$\cos(x) + \frac{1}{x}$$

`sec`

Syntax: `sec(`x`)`

The `sec` function returns the secant of the expression x where x is in radians.

seq Syntax: `seq(`$f(i), i = m..n$`)`

The `seq` command produces a **seq**uence of numbers or expressions based on the function f and the values m through n at which the function is evaluated.

Example Here is the sequence of the first 10 odd numbers.

```
> seq(2*i-1, i=1..10);
```

$$1, 3, 5, 7, 9, 11, 13, 15, 17, 19$$

Example Here is a sequence of squares starting with 25 and ranging to 100. Then we convert it into a list, L.

```
> a := seq(i^2, i=5..10);
```

$$a := 25, 36, 49, 64, 81, 100$$

```
> L := [a];
```

$$L := [25, 36, 49, 64, 81, 100]$$

showtangent Syntax: `showtangent(`$expression, x = a$`)`

Requires: `with(student)`

The `showtangent` command allows the user to plot the given *expression* (which should be in terms of the variable x) and its tangent line at the point $x = a$.

See Also: `student`

Example Here, **Maple V** plots the function $f(x) = 1 - x^2$ and its tangent at $x = \frac{1}{2}$.

```
> with(student):
> f := 1 - x^2;
```

$$f := 1 - x^2$$

```
> showtangent(f, x=1/2);
```

signum Syntax: `signum(`*expression*`)`

When the `signum` command is applied to a real *expression*, it returns 1 if the *expression* is positive, -1 if the *expression* is negative or 0 if the *expression* is 0.

simplify

Syntax: `simplify(`*expression*`)`

Syntax: `simplify(`$expression, \{equation_1, \cdots, equation_n\}$`)`

The `simplify` command is perhaps the most useful and the most confusing of the **Maple V** commands. Sometimes, the system's idea of "simple" is not the same as the user's idea, and confusion can occur.

The first syntax above is the standard use of the `simplify` command. It will try to make any simplifications to the given *expression* that it can. This involves combining like terms, simplifying trigonometric expressions, simplifying radicals, etc.

WARNING: Maple V may apply trigonometric identities when the `simplify` command is used and this may make the given expression more difficult to read.

$\boxed{\text{Example}}$ Here are various simplifications that **Maple V** can make.

```
> simplify(exp(ln(a) + 5*ln(x)));
```

$$ax^5$$

```
> f := (1+x)/x + (1-x)/x;
```

$$f := \frac{1+x}{x} + \frac{1-x}{x}$$

```
> simplify(f);
```

$$2\frac{1}{x}$$

The second syntax listed above can be used to specify relations that **Maple V** should try to apply to simplify the *expression*. The relations are given in the form of *equations*.

$\boxed{\text{Example}}$ We want **Maple V** to know that $x^3 + y^2 = 5$ in the following expression. Here is how we get it to do that. Note that the `subs` command doesn't work here.

```
> f := (x^3 + x^2 + y^2 + 1)/(z^4+1);
```

$$f := \frac{x^3 + x^2 + y^2 + 1}{z^4 + 1}$$

```
> simplify(f,x^3+y^2=5);
```

$$\frac{x^2 + 6}{z^4 + 1}$$

```
> subs(x^3+y^2=5,f);
```

$$\frac{x^3 + x^2 + y^2 + 1}{z^4 + 1}$$

simpson Syntax: `simpson(`$expression, x = a..b$`)`

Syntax: `simpson(`$expression, x = a..b, n$`)`

Requires: `with(student)`

The `simpson` command allows the user to approximate the integral of the given *expression* (which should be in terms of the stated variable x) between the limits a and b using Simpson's rule. The result is an unevaluated sum that can be evaluated using `value` or `evalf`.

The first syntax above assumes that the user wants four intervals. In the second syntax, the user can specify n intervals.

See Also: student, trapezoid

Example This example finds an approximation for the integral

$$\int_0^1 e^y\,dy$$

using Simpson's rule with $n = 16$. Compare with the result found with `int`.

```
> with(student):
> f := exp(y);
```

$$f := e^y$$

```
> s := simpson(f, y=0..1,15);
```

$$s := \frac{1}{48} + \frac{1}{48}e + \frac{1}{12}\left(\sum_{i=1}^{8} e^{1/8i-1/16}\right) + \frac{1}{24}\left(\sum_{i=1}^{7} e^{1/8i}\right)$$

```
> evalf(s);
```

$$1.718281975$$

```
> evalf(int(exp(y),y=0..1));
```

$$1.718281828$$

sin Syntax: `sin(`x`)`

The `sin` function returns the sine of the expression x where x is in radians.

`solve` Syntax: `solve(`$equation, x$`)`

Syntax: `solve(`$\{equations\}, \{x_1, x_2, \cdots, x_n\}$`)`

The `solve` command tries to find solutions for the given variable(s) in the given equation(s).

The first syntax above takes an *equation* and solves it for the variable x. There are various ways to call the `solve` command. Some are demonstrated below.

<u>**See Also**</u>**:** `fsolve`

Example Here show three variations of syntax that can be used to find the roots for the equation $x^3 + 216 = 0$. In

the first one, the entire syntax is as above. In the second one, **Maple V** assumes the expression is equal to zero since no equation is specified. Finally, in the last one, since the equation has only one variable, **Maple V** solves for the only available variable and assumes that the expression is set equal to zero.

```
> solve(x^3+216=0,x);
```

$$-6, 3 + 3I\sqrt{3}, 3 - 3I\sqrt{3}$$

```
> solve(x^3+216,x);
```

$$-6, 3 + 3I\sqrt{3}, 3 - 3I\sqrt{3}$$

```
> solve(x^3+216);
```

$$-6, 3 + 3I\sqrt{3}, 3 - 3I\sqrt{3}$$

The second syntax listed in this definition is the syntax for solving a system of equations in several unknowns.

Example In this example, we solve the system:

$$x + 3y - z = 17, y + z = 6, x^2 + z^2 = 10$$

for x, y, and z.

```
> eqns := x+3*y-z=17, y+z=6, x^2+y^2=10;
```

$$eqns := x + 3y - z = 17, y + z = 6, x^2 + z^2 = 10$$

```
> solve({eqns}, {x,y,z});
```

$$\{x = 3, y = 5, z = 1\}, \{x = \frac{-53}{17}, y = \frac{111}{17}, z = \frac{-9}{17}\}$$

sqrt Syntax: `sqrt(`x`)`

The `sqrt` command returns the square root of the expression x.

Example

```
> a := 1 + 2*I;
```

$$a := 1 + 2I$$

```
> b := sqrt(a);
```

$$b := \sqrt{1 + 2I}$$

```
> c := evalc(b);
```

$$c := \frac{1}{2}\sqrt{2 + 2\sqrt{5}} + \frac{1}{2}I\sqrt{-2 + 2\sqrt{5}}$$

stop Syntax: `stop`

The `stop` command exits from a **Maple V** session. It does not require a semicolon or a colon. It is equivalent to `done` or `quit`.

student Syntax: `with(student)`

The `student` package contains many useful commands for students learning calculus. Those covered by this dictionary are: changevar, completesquare, integrand, intparts, leftbox, leftsum, middlebox, middlesum, rightbox, rightsum, showtangent, simpson, trapezoid. A full list of `student` commands can be gotten with the line `> ?student`

See Also: `changevar, completesquare, integrand, intparts, leftbox, leftsum middlebox, middlesum, rightbox, rightsum, showtangent, simpson, trapezoid, with`

subs Syntax: `subs(`$equation(s), expression$`)`

The `subs` command allows the user to make **subs**titutions into the given *expression*. The substitutions are made via the *equations* supplied by the user.

Example Here the values $x = 1, y = 5$, and $z = -1$ are substituted into the expression $x^2 + y^2 + z$.

```
> f := x^2 + y^2 + z;
```

$$f := x^2 + y^2 + z$$

```
> subs(x=1, y=5, z=-1, f);
```

Note that in the above work, `f(1,5,-1)` would not have produced the desired result.

`sum, Sum`

Syntax: `sum(`$f(i), i = m..n$`)`

The `sum` command is used to produce the summation of terms described by the function $f(i)$ as i assumes values from m to n. The syntax above is equivalent

$$\sum_{i=m}^{n} f(i).$$

The `Sum` command can be used in the same manner to create an unevaluated sum. Use the `value` to evaluate a `Sum`.

See Also: `value`

Example Here **Maple V** finds the value of the sum of the numbers from 1 to 50.

```
> sum(i,i=1..50);
```

$$1275$$

Example **Maple V** can also find values of sums with variable upper limits. Here **Maple V** finds the sum of the first n numbers.

```
> sum(i,i=1..n);
```

$$\frac{1}{2}(n+1)^2 - \frac{1}{2}n - \frac{1}{2}$$

`tan`

Syntax: `tan(`x`)`

The `tan` function returns the tangent of the expression x where x is in radians.

`trapezoid`

Syntax: `trapezoid(`$expression, x = a..b$`)`

Syntax: `trapezoid(`$expression, x = a..b, n$`)`

Requires: `with(student)`

The `trapezoid` command allows the user to approximate the integral of the given *expression* (which should be in terms of the stated variable x) between the limits $x = a$ and $x = b$ using the trapezoid rule. The result is an unevaluated sum that can be evaluated using `value` or `evalf`.

The first syntax above assumes that the user wants four intervals. In the second syntax, the user can specify n intervals.

See Also: `simpson, student`

Example This example finds an approximation for the integral

$$\int_0^1 e^y \, dy$$

using the trapezoid rule with $n = 16$. Compare with the result found with `int`.

```
> with(student):
> f := exp(y);
```

$$f := e^y$$

```
> s := trapezoid(f, y=0..1,15);
```

$$s := \frac{1}{32} + \frac{1}{32}\left(\sum_{i=1}^{15} e^{1/16 i}\right) + \frac{1}{32} e$$

```
> evalf(s);
```

$$1.718841128$$

```
> evalf(int(exp(y),y=0..1));
```

$$1.718281828$$

`tutorial` Syntax: `tutorial()`

Syntax: `tutorial(n)`

The `tutorial` command runs the on-line tutorial. If an integer n is specified, the n^{th} tutorial is started.

`unapply` Syntax: `unapply(`$expression, x_1, x_2, \cdots, x_n$`)`

The `unapply` command creates a function from the given *expression* and makes it a function of the variables $x_1, x_2, \cdots, x_n$.

Example The expression f below is made into a function of x.

```
> f := sin(x)/x;
```

$$f := \frac{sin(x)}{x}$$

```
> f1 := unapply(f,x);
```

$$f1 := x \to \frac{sin(x)}{x}$$

```
> f1(0);
```

$$1$$

usage Syntax: `usage(`*command*`)`

The `usage` command returns a template for the calling sequence of the given *command*. This information is grabbed from the help page for the *command* under the heading `CALLING SEQUENCES`. The `usage` command is equivalent to `??`.

value Syntax: `value(`*expression*`)`

The `value` command forces evaluation of unevaluated expressions like sums, limits, and integrals.

$\boxed{\text{Example}}$ Below an unevaluated sum, integral, and limit are evaluated with `value`.

```
> S := Sum(1/k^2, k=1..infinity);
```

$$S := \sum_{k=1}^{\infty} \frac{1}{k^2}$$

```
> value(S);
```

$$\frac{1}{6}\pi^2$$

```
> A := Int((x+1)/x,x=2..5);
```

$$A := \int_2^5 \frac{x+1}{x} dx$$

```
> value(A);
```

$$3 + \ln(5) + \ln(2)$$

```
> L := Limit(sin(x)/x, x=0);
```

$$L := \lim_{x \to 0} \frac{\sin(x)}{x}$$

```
> value(L);
```

$$1$$

while Syntax: `while` *condition* `do` *statements* `od`

The `while` command forms a loop with the `do ... od` statements. The loop will execute the *statements* as long as the *condition* holds true. The *condition* is typically some relation like $x < y$, $x > y$, etc.

<u>**See Also**</u>**:** `break, do, for, next`

with Syntax: `with`(*package*)

The `with` command loads in a *package*, where the *package* is a set of commands that are not normally part of the **Maple V** command set. Examples of packages are `plots` and `student`.

Example Here is an example of the with command.

```
> with(plots);
```

[animate, animate3d, conformal, contourplot, cylinderplot, densityplot, display , display3d, fieldplot, fieldplot3d, gradplot, gradplot3d, implicitplot, implicitplot3d, loglogplot, logplot, matrixplot, odeplot, pointplot, polarplot , polygonplot, polygonplot3d, polyhedraplot, replot, setoptions, setoptions3d, spacecurve, sparsematrixplot, sphereplot, surfdata, textplot, textplot3d, tubeplot]

A complete list of packages can be found with `?packages`.

If memory is a consideration, the `with` statement may be modified to load a single command from a package. Also, the `with` statement can be avoided by calling a function in square brackets preceded by the package name.

Example Here are two ways to call the `middlesum` command from the `student` package without calling the entire package into memory.

```
> student[middlesum](sin(x),x=0..Pi);
> with(student, middlesum):
> middlesum(sin(x),x=0..Pi);
```

<u>**See Also**</u>**:** `plots, student`

zip Syntax: `zip`($f, list_1, list_2$)

Syntax: `zip`($f, list_1, list_2, default$)

The `zip` command is used to combine two lists with a function f that takes two arguments (one from each list) and performs some operation with them. The first syntax above combines two lists of equal length with a function f. The second syntax allows for lists of different lengths and lets $default$ be the value from the shorter list when it runs out of values to match with the longer list. Below are examples.

$\boxed{\text{Example}}$ Below we combine two lists of equal length into a list of ordered pairs.

```
> List1 := [1,2,3,4];
```

$$List1 := [1, 2, 3, 4]$$

```
> List2 := [2,3,6,9];
```

$$List2 := [2, 3, 6, 9]$$

```
> f := (x,y) -> [x,y];
```

$$f := (x, y) \to [x, y]$$

```
> List3 := zip(f,List1,List2);
```

$$List3 := [[1, 2], [2, 3], [3, 6], [4, 9]]$$

$\boxed{\text{Example}}$ Below we combine two lists of different lengths into a list which has as its elements the product of corresponding elements from the two lists.

```
> List1 := [1,2,3,4,5,6];
```

$$List1 := [1, 2, 3, 4, 5, 6]$$

```
> List2 := [4,6,8,10];
```

$$List2 := [4, 6, 8, 10]$$

```
> f := (x,y) -> x*y;
```

$$f := (x, y) \to xy$$

```
> List3 := zip(f,List1, List2, Pi);
```

$$List3 := [4, 12, 24, 40, 5\pi, 6\pi]$$