Maple for Math Majors

Roger Kraft
Department of Mathematics, Computer Science, and Statistics
Purdue University Calumet
roger@purduecal.edu

2. Variables, Assignment, and Equations

2.1. Introduction

In this worksheet we look at how Maple treats variables and at Maple's rules for how variables can be named. We also look at how variables are given values with the assignment operator, and we compare Maple's use of the assignment operator and the equal sign with the standard mathematical use of the equal sign in equations.

>

2.2. Assigned and unassigned variables

Every variable in Maple will have one of two states; either it is an assigned variable, or it is an unassigned variable.

Assigned variables are names for some value. In other words, an assigned variable is a name that represents something. An assigned variable can represent almost anything, a number, an expression, a function, an equation, a graph, a solution, a list of things, another variable, etc. Assigned variables are sometimes called "programming variables" because they act, more or less, like the variables in traditional programming languages. Assigned variables are also sometimes referred to as a "label for a result" or simply as "labels".

Unassigned variables are names that do not yet represent a particular value. Unassigned variables are sometimes called "free variables", because they are free to take on any value. They are also sometimes called "unknowns" because they do not have a value. Other terms used as synonyms for "unassigned variables" are, mathematical variables, mathematical symbols, mathematical unknowns, algebraic unknowns, and indeterminates.

All variables begin their life in Maple as unassigned variables. To change an unassigned variable to an assigned one, we use the **assignment operator**, which is a colon followed by an equal sign (i.e. **:=**). A Maple command with an assignment operator in it is called an **assignment statement**.

Here are some examples of assigned and unassigned variables. Until you give the variable **x** a value, **x** is an unassigned variable. We sometimes say that an unassigned variable represents itself.

> **x;**

Let us change **x** into an assigned variable.

```
> x := 3;
```

Now **x** represents the integer 3. (Sometimes we will say "the value of **x** is 3". Other times we might say "**x** is a name for 3", or "**x** refers to 3", or "**x** has the value 3", or "**x** is a label for 3", or "3 is assigned to **x**".)

```
> x;
```

Here is how we change **x** back into an unassigned variable. We assign to **x** a right quoted copy of **x**. (Right quotes are the single quotes found on the right hand side of the keyboard.)

```
> x := 'x';   # Those are both right quotes.
```

Now **x** represents itself again (or, we can say that **x** no longer has a value).

```
> x;
```

Converting an assigned variable back into an unassigned variable is an important, and very common, operation in Maple. We will refer to this operation as **unassigning** a variable. It may seem odd to use an assignment statement to unassign a variable. It helps to think of unassigned variables as variables whose value is themselves (as opposed to thinking of them as variables that do not have a value).

```
>
```

In the following command, **p** becomes an assigned variable and **x** is remains an unassigned variable.

```
> p := x^2+2*x+1;
```

Now we can see that **p** represents the polynomial **x^2+2*x+1**.

```
> p;
```

Let us change **x** into an assigned variable.

```
> x := 3;
```

Now what will **p** evaluate to?

```
> p;
```

The last example brings up the idea of **evaluation rules**. How should Maple have evaluated **p**? It would have been reasonable to return either **x^2+2x+1** or **16**. But Maple uses an evaluation rule called **full evaluation**, i.e., it evaluates everything it can, until there are only numbers and unassigned variables left, which is how Maple arrived at the value 16. But even though **p** *evaluates* to 16, 16 is not the value that is *assigned* to **p**. The value assigned to **p** is still the polynomial **x^2+2*x+1**. We can see that by making **x** unassigned again, and then seeing what **p** evaluates to.

```
> x := 'x';
```

```
> p;
```

It is important to remember that *what value a variable has assigned to it and what value a variable evaluates to may be different*.

Here is another example. Let **p** represent an expression in the unassigned variables **a**, **b**, **c**, and **t**.

```
> p := a*t^2+b*t+c;
```

Now change **t** into an assigned variable.

```
> t := 3;
```

Here is what **p** evaluates to now (thought the valued assigned to **p** has not changed, it is still

`a*t^2+b*t+c`).

```
> p;
```

Now change **b** into an assigned variable.

```
> b := 5;
```

When we ask Maple for the value of **p**, Maple evaluates everything it can.

```
> p;
```

We will discuss evaluation rules in more detail in a later worksheet.

```
>
```

Exercise: Check that the value assigned to **p** is still `a*t^2+b*t+c`.

```
>
```

In this next example, **x** is an assigned variable and its value is **y**, an unassigned variable.

```
> x := y;
```

If we use the variable **x** in an expression, then the **x** will evaluate to **y** in the expression.

```
> x^2-3*x+exp(x);
```

If we now make **y** an assigned variable with the value 2, then **x** will evaluate to 2 also.

```
> y := 2;
```

```
> x;  # Now x evaluates to 2 also.
```

It is important to make a distinction here. The variable **x** does not have the value 2 *assigned* to it, rather, the variable **x** *evaluates* to the value 2. The value assigned to **x** is still **y** (**x** is really a name for **y** and **y** is a name for 2). Let us unassign **y**.

```
> y := 'y';
```

Now **x** once again evaluates to **y**.

```
> x;
```

We will say more later about the distinction between the value assigned to a variable and what a variable evaluates to.

```
>
```

Here is an example that may seem unusual. We will make **w** (for" **w**eird") an assigned variable whose value is an expression in the unassigned variable **hello**.

```
> w := hello^2+2*hello-3;
```

As far as Maple is concerned, this expression is no different than any other expression. We can factor it.

```
> factor( w );
```

We can turn it into an equation and solve the equation.

```
> solve( w=0, {hello} );
```

Now let **x** represent the name **hello**.

```
> x := hello;
```

In the next command, the command looks normal, but the output looks unusual.

```
> polynomial := x^2+2*x-3;
```

```
> factor( polynomial );
```

These examples are meant to show that Maple does not care what kind of variable names you use. But we are very, very accustomed to the idea of using single letter variable names (like **a**, **b**, **c**, **x**, **y**, **z**, **t**) in math formulas. In general, most Maple users use single letter variable names in mathematical expressions, but use word-like variable names to label an expression or an equation. For example we can use **quad** as a name for a quadratic expression in **t**.

```
> quad := a*t^2+b*t+c;
```

We are not likely to use **x** as a name for an expression in the unknown **bozo**.

```
> x := 3*bozo+bozo^2*exp(bozo);
```

We can use the names **eqn1** and **eqn2** for a pair of simultaneous linear equations in the unknowns **u** and **v**.

```
> eqn1 := 3*u+2*v=0;
> eqn2 := 5*u-7*v=1;
```

Now we can easily refer to these two equation in, for example, a **solve** command.

```
> solve( {eqn1,eqn2}, {u,v} );
```

And we can use **soln** as a convenient name for the solution of these equations.

```
> soln := %;
```

Remember, using word-like variable names (or labels) for things like expressions or equations, and using single letter variable names (or unknowns) in the expressions or equations, is just a convention. Maple itself does not care what you call anything

```
>
```

Let us consider our last example some more. Notice how efficient Maple is. Maple uses the same assignment operator for such seemingly different tasks as giving a variable a value, as in **x:=5**, and giving an equation a label, as in **eqn1:=3*u+2*v=0**. To get a sense of the difference between these two concepts, consider how equations are labeled (i.e., named) in mathematics books. This is how a math book would present and label the last two equations given above.

$$3\,u + 2\,v = 0 \qquad\qquad (1)$$
$$5\,u - 7\,v = 1 \qquad\qquad (2)$$

The first equation can now be referred to as Equation (1) and the second equation as Equation (2). From this point of view, labeling an equation has nothing to do with assigning a value to a variable! One of the great design features of Maple is that any variable can take on any kind of value and a single assignment operator is used for associating any value to any variable. So a variable can represent an expression or an equation just as easily as it can represent a number.

Here is another example of this kind of efficiency. The next three commands give names to graphs.

```
> g1 := plot( [[0,0], [1,2], [2,0]], color= red );
> g2 := plot( [[0,2], [1,0], [2,2]], color=green );
> g3 := plots[textplot]( [1,1,"My favorite graph."], color=blue
    );
```

The variables **g1**, **g2**, and **g3** have graphs as their values. Now we can refer to these graphs in a

display command, which will combine them into one graph and then display the combined graph.

```
> plots[display](g1,g2,g3);
```

Because a graph can be the value of a variable, Maple allows us to refer to and manipulate whole graphs much like we would refer to and manipulate numbers, expressions, or equations. For example, the next command uses **subs** to change the color used in **g2** from green to black.

```
> g2 := subs( (RGB,0.,1.00000000,0.)=(RGB,0,0,0), g2 );
> plots[display](g1,g2,g3);
```

In Maple, we can label and manipulate *anything* using just a few basic commands.

```
>
```

Lastly, here is an interesting feature of Maple's assignment operator. We can use the assignment operator to assign more than one variable at a time.

```
> x, y := 1, 2;
> x, y;
```

We need to have as many values on the right hand side of the assignment operator as we have variables on the left hand side. So the next command does not set both **x** and **y** to zero.

```
> x, y := 0;
```

We can use this trick to unassign more than one variable in a single command.

```
> x, y := 'x', 'y';
```

We will use this feature of the assignment operator only when we need to assign large numbers of "automatically generated" names at one time. See the section on concatenated names below.

```
>
```

Exercise: Is the following Maple command

```
> x, y := a, b;
```

equivalent to the following execution group?

```
> x := a;
> y := b;
```

Hint: Consider the case where initially **a:=y**, **b:=x**, **x:=1**, and **y:=2**.

```
>
```

```
>
```

2.3. Equal signs, equations, and assignment

Before going any further, let us use the **restart** command to return all the assigned variables that we have used back into unassigned variables.

```
> restart;
```

In standard mathematical notation, if we want to express the idea that the variable a is suppose to represent the number 5, we would write the formula $a = 5$. What if we use this formula as a Maple command?

```
> a = 5;
```

Did this command make the variable **a** represent the number 5?

```
> a;
```

No, **a** is still an unassigned variable. In Maple, if we want the variable **a** to represent the number 5, we need to use an assignment statement and make **a** an assigned variable.

```
> a := 5;
> a;
```

In standard mathematical notation, if we want to express the idea that we should take the expression $x^2 - 2x - 1$ and find values for x that makes this expression equal to zero, then we would write the formula

$$x^2 - 2x - 1 = 0.$$

Notice that in this formula the equal sign does not have the same meaning as in the formula $a = 5$. The formula $a = 5$ means that a is a name for the number 5, but the formula $x^2 - 2x - 1 = 0$ does *not* mean that $x^2 - 2x - 1$ is a name for the number 0. The formula $x^2 - 2x - 1 = 0$ can be translated directly into Maple.

```
> x^2-2*x-1 = 0;
```

And now we can ask Maple to solve it for us.

```
> solve( %, x );
```

We see from these examples that the equal sign (=) in mathematics can have at least two different meanings. Maple chooses to have two different symbols for these two different meanings. The equal sign in a mathematical formula can be translated in a Maple command into either the Maple assignment operator (**:=**) or the Maple equal sign (**=**). We know that the assignment operator in Maple gives a value to a variable. What does an equal sign do in Maple?

Exercise: In the following Maple command, what role does the equal sign play?

```
> a := b = 2;
```

In Maple, equal signs are used to make equations. An **equation** in Maple is an equal sign, **=**, with an expression on either side of it. Here are some examples of equations.

```
> b = 2;
> x^2-3*x+2 = 5*x^2+sqrt(2)*x+Pi;
> 5*u-7*v = 0;
> y = 12*exp(z)*sin(t);
```

So the Maple command **a:=b=2** assigns the (simple) equation **b=2** as the value of the variable **a**. Another way to put it is that the command **a:=b=2** makes the variable **a** a name (or a label) for the equation **b=2**.

```
>
```

So then, what is the purpose of an equation in Maple? What are equations used for? What does the Maple command **b=2** do?

For the most part, an equation in Maple is used to ask a question; is the left hand side of the equation

equal to the right hand side? The answer will be either yes or no. If the answer is yes, then we say that the equation is **true**; if the answer is no, then we say that the equation is **false**. We can ask Maple if an equation is true or false by using the **evalb** command. (**evalb** is an abbreviation of " **eval**uate **b**oolean". Booleans are expressions that are either true or false. There are other kinds of boolean expression besides equations. We will say more about booleans in a later worksheet.)

Here are a few examples of using **evalb** to ask if an equation is true or false.

```
> evalb( 0=1 );
```

Notice that, regardless of what you may have been taught in grade school, there is nothing wrong with writing down the equation $0 = 1$. It is simply a false equation. Some equations are true and some are false. Here is another false one.

```
> evalb( 2+2=5 );
```

Again, there is nothing improper with wring down the equation $2 + 2 = 5$, it just happens to be a *false* equation (and we will see later that false equations can be just as useful as true equations). Here is a true equation.

```
> evalb( 2+2=4 );
```

Here is another true equation.

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

This equation is true even though **x** is an unassigned variable since no matter what value **x** may end up having, it is always the case that **x** equals itself. On the other hand, the following equation is false.

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

This equation is false because both **x** and **y** are unassigned variables and the name **x** is not the same as the name **y**. But if we let **x** and **y** both represent -2, then **x=y** becomes a true equation.

```
> x := -2;
> y := -2;
> evalb( x=y );
```

Here is another example.

```
> x := 27;
> y := -16;
> evalb( 3*x+5*y=1 );
```

The equation is true because the values 27 and -16 for **x** and **y** make the left and right hand sides of the equation **3*x+5*y=1** equal.

```
>
```

If an equation has an unassigned variable in it, then the equation can be used to ask a more sophisticated question: for which values of the unassigned variable is the left hand side of the equation equal to the right hand side, or to put it another way, for which values of the unassigned variable is the equation true. Finding the values of the unassigned variable that make an equation true is called **solving the equation**. For example, the equation $3\,a + 1 = 0$ is solved by giving a the value $-1/3$ and the equation $x^2 - 1 = 0$ is solved by giving x either the value 1 or the value -1. Maple has the **solve** command for solving equations with unassigned variables in them.

```
> x := 'x':  # Make sure x is unassigned.
> solve( x^2-1=0, x );
> a := 'a':  # Make sure a is unassigned.
> solve( 3*a+1=0, a );
```
We tell the **solve** command what equation to solve and what unassigned variable to solve for. We can check that a solution makes an equation true using the **subs** command.
```
> subs( a=%, 3*a+1=0 );
```
Here is another example.
```
> u, v := 'u', 'v':  # Make sure both u and v are unassigned
> solve( 3*u-v=0, {u} );
```
Now we see why the **solve** command needs to be told which variable to solve for since there may be more than one unassigned variable in an equation. In the last example, Maple solved for **u** in terms of **v**.
```
>
```

Exercise: Modify the last command so that Maple solves the equation for **v** in terms of **u**.
```
>
```

Besides asking a question, there is another way that Maple can use an equation. Consider the following example. Here we use the **solve** command to solve a simple equation.
```
> x := 'x':  # Make sure x is unassigned.
> solve( 5*x+2=0, x );
```
So the value $-2/5$ for **x** solves this equation. Now let us re-enter this **solve** command in a slightly different way.
```
> solve( 5*x+2=0, {x} );
```
Notice that we put braces around the last **x** in the **solve** command and then the **solve** command gave us our answer as an "equation" (inside of braces). So here an equation is being used to display a solution. The solution "equation" tells us that when **x** is $-2/5$ the original equation is true. But the equation in the solution is *not* an assignment, so **x** is still an unassigned variable.
```
> x;
```
The equation in the answer to the **solve** command did not "do" anything. The equation was just a convenient way to display the solution of the equation. So besides being used to ask questions, equations are sometimes used in Maple to display answers.
```
>
```

But Maple has many tricks up its sleeves. There is a way to make the equation in the solution to the last **solve** command "do" what you think it should do, which is make **x** represent $-2/5$. Here is the **solve** command again.
```
> solve( 5*x+2=0, {x} );
```
Remember, at this point **x** is still unassigned. But now we can use the **assign** command to tell Maple to treat the equation in the last answer as an assignment.
```
> assign( % );
```

Notice that the **assign** command is one of the few Maple commands that does its work silently, so there was no output from **assign**. But what it did was take the equation **x=-2/5** and treat it as an assignment. So now **x** is an assigned variable.

```
> x;
```

Here is another example.

```
> x, y := 'x', 'y':  # Make sure x and y are unassigned.
> solve( {3*x-5*y=2, x+9*y=0}, {x,y} );
```

At this point both **x** and **y** are still unassigned variables. But the next **assign** command will change that.

```
> assign( % );
> x, y;
```

In general, if an equation has an unassigned variable on its left hand side, then the **assign** command can be used to make the equation act like an assignment. Here is an example.

```
> p := 'p':  # Unassign p and x.
> x := 'x':
```

Put **p** on the left hand side of an equation.

```
> p = x^3 + x^2 + x + 1;
```

p is still an unassigned variable.

```
> p;    # Still unassigned.
> assign( %% );
> p;    # Now p is assigned.
>
```

Here is a way that an equal sign is used in Maple as a sneaky kind of assignment. In a **subs** command we tell Maple to plug a certain value in for a certain variable in a certain expression, like this.

```
> subs( x=1/3, x^2-x+1 );
```

The equation **x=1/3** acts like an assignment *for just that command*. The variable **x** is still unassigned.

```
> x;
```

We will see a few other places where Maple uses an equal sign as a kind of sneaky assignment. But you should remember that in general, in Maple,

> *An equal sign is used in equations to ask a question.*
> *The assignment operator is used to change the value of a variable.*

```
>
```

We have seen that in Maple an equal sign can be used either to ask a question in an equation or, in a certain sense, to represent an assignment. In Maple, determining when an equal sign is meant to do the former as opposed to the later is usually pretty clear. There are only a few places in Maple where an equal sign is meant to represent a kind of assignment. But in standard mathematical notation, it can sometimes be tricky to distinguish these two uses of an equal sign (i.e., asking a question in an

equation vs. assigning a value to a variable). In fact, standard mathematical notation by itself usually *cannot* distinguish between these two uses of an equal sign. Close reading of the context that the equal sign appears in will usually be required.

Here is an example of the kind of ambiguity that can arise from the use of equal signs in standard mathematical notation. In mathematics books you often see formulas like the following

$$f(x) = x^2 - 2x - 1$$
$$f(x) = 0$$

How are we to interpret these two formulas? If the first one is defining the function f, then is the second one redefining f? Or is the second one "setting the function f to zero" as in "find the values of *x* that make the function f equal to zero"? The mathematical notation is ambiguous about these two possible interpretations of the second formula. In Maple, things are more clear. If we take the first formula to be the definition of the function **f** and we take the second formula to be an equation, then we would translate the two formulas into a Maple assignment statement and a Maple equation.

```
> f := x^2-2*x-1;
> f = 0;
```

Now we can use the **solve** command to solve the equation and find the roots of **f**.

```
> solve( %, x );
```

But if instead we interpret the first formula to be the definition of **f** and the second formula as a redefinition of **f**, then we would translate both of the formulas into assignment statements.

```
> f := x^2-2*x-1;
> f := 0;
```

Now **f** is a name for zero instead of being a name for **x^2-2*x-1**. Notice that there is no ambiguity in either of the Maple translations of the mathematical formulas.

We will return several times in later worksheets to this distinction between equations and assignments and to the ambiguity of the equal sign in standard mathematical notation.

```
>
```

Here is another example of how the assignment operator in Maple is not quite like an equal sign in standard mathematical notation. Let us give the variable **y** a value.

```
> y := 3;
```

Now consider the following assignment statement.

```
> y := y+1;
>
```

From the output we see that the statement gave **y** the value 4. But the statement really did something more interesting than that. The assignment **y:=y+1** tells Maple to give **y** a new value that is one more than the current value of **y**. Since the value of **y** before the statement was 3, the value of **y** after the statement is 4. (Go back and re-execute the last assignment statement several times to see what happens.) What if we try to translate this assignment statement into standard mathematical notation? If we use an equal sign in place of the assignment operator, we get the following formula.

$$y = y + 1$$

But elementary algebra would tell us to immediately cancel a y from both sides of the equal sign and leave us with $0 = 1$ which is a false equation. So the mathematical equation $y = y + 1$ is not at all equivalent to the assignment statement `y:=y+1`. In fact, there is no way to use standard mathematical notation to express the idea that the variable y should take on the value that is one more than its current value. So here we see a use of the assignment operator in Maple that does not have an equivalent use of an equal sign in mathematical notation (as opposed to the assignment `y:=3`, which would have the same meaning as $y = 3$).

As we will see in many of these worksheets, statements like `y:=y+1` are very common when writing programs. Since there is no mathematical equivalent to this kind of statement, some students find them awkward at first. But after a while, this programming idiom becomes very natural.

```
>
```

Exercise: Let `x` be an unassigned variable and suppose the `y` is assigned.

```
> x := 'x';
> y := 1;
```

What will repeated execution of the following assignment do? Explain why.

```
> y := y + x;
>
```

Exercise: Let `s` represent 0.

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

What will repeated execution of the following assignment do? Explain why.

```
> s := s,s;
>
```

One last comment about the difference between the equal sign and the assignment operator. Notice that the character `=` is symmetric but the character combination `:=` is not. This should help you to remember that equations are symmetric (the relational operator `=` is commutative) but assignment statements are not (the assignment operator `:=` is not commutative). In other words

```
> b = 5;
```

and

```
> 5 = b;
```

mean exactly the same thing as equations, but

```
> b := 5;
```

and

```
> 5 := b;
```

are not equivalent statements. In fact the second one is an error since you cannot assign a value to a number (numbers cannot be names).

It is interesting to note that in the C family of programming languages (C, C++, C#, and Java), the assignment operator is = and equations are denoted by == (double equal signs). The use of = as the

assignment operator obscures the fact that assignment is not commutative. The use of colon-equal as an assignment operator comes from the Algol and Pascal programming languages.

```
[ >
```

```
[ >
```

2.4 Equations and Solution Sets

In the last section we said that

> *an **equation** in Maple is an equal sign,* **=***, with an expression on either side of it*.

If there are no unassigned variables in either of the two expressions, then the equation asks a question: is the value of the left hand expression equal to the value of the right hand expression? If the values of these two expressions are equal, we say that the equation is true, if the values are not equal, we say that the equation is false.

```
[ >
```

If an equation has unassigned variables in it, then the equation can be used to ask a more subtle question. We can ask: for what values of the unassigned variables is the equation true and for what values of the unassigned variables is the equation false? For example, for a simple equation like

$$5\,x + 1 = 11,$$

if we assume that x is unassigned, then the value of 2 for x makes the equation true and any other value for x makes the equation false. For a slightly more complicated equation like

$$x^2 - 5\,x + 6 = 0,$$

there are two values for the unassigned variable that make the equation true, 2 and 3, and all other values make the equation false.

```
[ >
```

Here are two extreme examples. The equation

$$x + 1 = x$$

is not true for any value of the unassigned variable x (no number is one more than itself). On the other hand, the equation

$$x + 1 = \frac{2 + 2\,x}{2}$$

is true for all values of the unassigned variable x.

Here is a tricky example. Consider the equation

$$x^2 + 4 = 0.$$

We might be inclined to say that there is no value for the unassigned variable x that makes this equation true. And that would be correct, if we do not allow ourselves to us complex numbers. But if we allow ourselves to make use of the complex imaginary number $i = \sqrt{-1}$, then we can say that there are two values for x that make the equation true, $x = 2\,i$ and $x = -2\,i$. This example brings up an important point, how many solutions an equation has can depend on what kinds of values we are

allowed to plug into the unassigned variables. In the case of the equation $x^2 + 4 = 0$, there are no solutions if we can only use real numbers as the values for x, but there are two solutions if we can use complex numbers as the values for x.

⌈ >

Let us make two definitions.

Given an equation with unassigned variables in it, the **universal set** for the equation is the set of values that can be plugged into the unassigned variables.

Given an equation with unassigned variables in it and a universal set for the unassigned variables, the set of all values (from the universal set) that make the equation true is called the **solution set** of the equation (though it really should be called the "solution *subset* of the universl set").

Be sure to keep straight the difference between an equation's universal set and its solution set. The universal set is all values that we are *allowing* to be plugged into the unassigned variables. Some of the values from the universal set will make the equation true, and some values will make the equation false. The solution set is those values from the universal set that make the equation true. Notice that these definitions show us how important "false" equations are. The solution set of the equation $x - 3 = 1$ (when the universal set is all real numbers) is the set made up of the single number 4. We know that, say 3, is not in this solution set because the equation $0 = 1$ is false. We need false equations, like $0 = 1$, to tell us which points from a universal set are *not* in a solution set.

Exercise: What is the solution set for the equation $x + 1 = x$? What is the solution set for the equation $x + 1 = \dfrac{2 + 2\,x}{2}$? (Be sure to specify what your universal set is.)

⌈ >

We have already seen that Maple can solve equations using the **solve** command. When you use the **solve** command, the universal set is always considered to be all the complex numbers. So if we ask Maple to solve the equation $x^2 + 4 = 0$, it gives us two solutions.

⌈ > **solve(x^2+4=0, x);**

If we ask Maple to solve the equation $x^3 - 8 = 0$, it will give us three solutions.

⌈ > **solve(x^3-8=0, x);**

And if we ask Maple to solve the equation $x^3 - 2\,x^2 - 5\,x + 10 = 0$, it will give us three solutions.

⌈ > **solve(x^3-2*x^2-5*x+10=0, x);**

What if we wanted to solve these equations using a different universal set? Maple has some ability to let us change the universal set.

Here is how we can solve an equation with the universal set being the real numbers.

⌈ > **RealDomain[solve](x^3-8=0, x);**

Here is how we can solve an equation with the universal set being the integers.

```
> isolve( x^3-2*x^2-5*x+10=0, x);
>
```

Exercise: Consider the equation

$$x^5 - 2\,x^4 - 25\,x + 50 = 0.$$

Use appropriate Maple commands to show the answer to each of the following questions.
What is the solution set if the universal set is the integers?

```
>
```

What is the solution set if the universal set is the real numbers?

```
>
```

What is the solution set if the universal set is the complex numbers?

```
>
```

Exercise: Use Maple to determine what the solution set is to the equation $x + 1 = x$. What kind of answer does Maple give? Why?

```
>
```

Exercise: Use Maple to determine what the solution set is to the equation $x + 1 = \dfrac{2 + 2\,x}{2}$. What kind of answer does Maple give? Why?

```
>
```

We have been looking at ways of changing the solution set of an equation by changing what its universal set is. Here is another, and very important, example of how the solution set of an equation can be changed by changing the equation's universal set.

The simple equation $x = 3$ has only one number in its solution set (if we assume that the solution set is the set of real numbers). But you may also remember that $x = 3$ can be considered the equation of a vertical line in the plane. Here is how we can get Maple to graph this vertical line.

```
> plots[implicitplot]( x=3, x=-5..5, y=-5..5, view=[-5..5, -5..5]
  );
```

How do we reconcile these two approaches to $x = 3$, the idea that the solution set of $x = 3$ is the single number 3, but the graph of $x = 3$ is a line? We reconcile them by saying that we have one equation, $x = 3$, but two different universal sets for that equation. In one case, the universal set is the (one-dimensional) number line of all real numbers. In the other case, the universal set is the (two-dimensional) *xy*-plane of all ordered pairs of real numbers.

Let us consider the case of the equation $x = 3$ with the universal set being all points (x, y) from the two-dimensionall plane. Does the point (3,5) make the equation $x = 3$ true? Yes, since we take the number 3 and plug it into x in the equation and get the true equation 3=3. Similarly for the point (3,-237). In fact, for any point $(3, y)$ from the plane, that point will make the equation $x = 3$ true. The

specific value of *y* does not matter. But all possible points of the form (3, *y*) make up a vertical line passing through the point (3,0) on the *x*-axis. Notice also that the point (5,3) *does not* make the equation *x* = 3 true (Why? Hint: The order of the numbers in (5,3) is important.). So now we can say that the solution set of the equation *x* = 3 is exactly the vertical line through the point (3,0) (*if* we agree that the universal set is the plane).

⌈ >

But now I am going to claim that the solution set of the equation *x* = 3 is a vertical plane in three-dimensional space. Here is how I can get Maple to verify this claim.

⌈ > **plots[implicitplot3d](x=3, x=-5..5, y=-5..5, z=-5..5,**
    axes=normal);

How do I explain this new solution set? By specifing a new universal set. In this case, the universal set for the equation *x* = 3 is the three-dimensional *xyz*-space of all ordered triples of the form (*x*, *y*, *z*). Notice how any point in space of the form (3, *y*, *z*) will make the equation *x* = 3 true (the specific values of *y* and *z* do not matter). But all possible points of the form (3, *y*, *z*) make up a vertical plane passing through the point (3,0,0) on the *x*-axis (this plane is parallel to the *yz*-plane).

⌈ >

So now we have three different universal sets for the equation *x* = 3 and each universal set gives us a different solution set for this equation.

Notice how we are not changing what kind of number *x* can represent (when, as before, we changed the universal set for an equation from integers to real numbers, and then to complex numbers). Here, we are changing the dimension of the universal set from the one-dimensional *x*-axis to the two dimensional *xy*-plane, to the three-dimensional *xyz*-space. In each of these universal sets, *x* can represent any real number.

Exercise: Consider the equation $x^2 - 9 = 0$ with the universal set being the *x*-axis. What is the solution set? Explain why your answer is correct. Use a Maple command to demonstrate that your answer is correct.

⌈ >

Exercise: Consider the equation $x^2 - 9 = 0$ with the universal set being the *xy*-plane. What is the solution set? Explain why your answer is correct. Use a Maple command to demonstrate that your answer is correct.

⌈ >

Exercise: Consider the equation $x^2 - 9 = 0$ with the universal set being *xyz*-space. What is the solution set? Explain why your answer is correct. Use a Maple command to demonstrate that your answer is correct.

⌈ >

```
>
```

2.5. Variable names

In Maple, variables are really called **names**. Maple has two kinds of names, **symbols** and **indexed names**. In this section we define what a symbol is and in the next section we define indexed names.

In Maple, almost any string of characters can be a variable name (i.e., a symbol). The precise rule is that a name must begin with a letter or an underscore and can be followed by any number of letters, digits, and underscores. Here are some examples of valid variable names.

```
> xyz, XYZ, x_y_z, _x, x1, x2, x3, hello_there;
```

Since names must begin with a letter (or an underscore) something like `1x` cannot be a name and, more importantly, numbers cannot be names.

```
>
```

Two important notes. First, Maple names are case sensitive, so `apple` is a different from `Apple`. Second, Maple names that begin with an underscore, like `_Z`, are used as special names internally to Maple. If you start creating your own variable names that begin with an underscore, Maple may start to behave strangely. So *do not* create your own variable names that begin with an underscore.

Besides names that begin with an underscore, there are certain other names that you should not (or even cannot) use in Maple. For example Maple has a list of 46 keywords that you cannot use as names. Maple also has reserved a number of names for predefined variables like `Pi`; these names are referred to as initially known names. You could, if you really wanted to, use one of these names for your own variable, but that it is not a good idea. Maple also has a list of initially known function names that you should shy away from redefining. If you should ever try to use a name that Maple has reserved for itself, here are the kinds of error messages that you will get.

```
> gamma := 0;
```

This is the standard error message that you get when you try to redefine one of the reserved names that Maple has "protected". On the other hand, the following error message is pretty cryptic.

```
> I := 0;
```

`I` is a reserved name but it is not "protected", so that is not why we got the error message. We got the error message because `I` is an **alias**, which means that `I` is just a short hand for something else. In the case of `I`, it is a short hand for `(-1)^(1/2)`. So the assignment `I:=0` really means `(-1)^(1/2):=0`, which explains the error message, since you cannot assign a value to a number.

```
>
```

While we are on the subject of Maple's predefined names, one surprising fact about Maple is that the important mathematical constant **e** (the base for the natural logarithm) does *not* have a predefined name built into Maple. To get this constant you need to enter the following expression.

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

Notice how confusing this can be. Maple typeset the expression **exp(1)** into Standard Math Notation as a letter e, but the name **e** is *not* a name for **exp(1)**. The name **e** is unassigned.

```
> e;
> evalf( e );
```

Look carefully at the following output.

```
> exp(1) * e;
```

The expression **exp(1)** is displayed in Standard Math Notation in a different font than the name **e**. An easy mistake to make is to read a formula like the following

$$\mathbf{e}^x \cos(x)$$

and translate it into Maple as

```
> e^x * cos(x);
```

which looks good, but is wrong!

```
>
```

One class of variable names that you are free to use but have special significance are the letters from the Greek alphabet. If you spell out the name of a Greek letter in Maple Notation, it will appear in Standard Math Notation typeset as a Greek letter. Here are some upper case Greek letters.

```
> Alpha, Beta, Gamma, Theta, Omega;
```

Here are some lower case Greek letters.

```
> alpha, beta, gamma, theta;
```

It is important to realize that when you enter **theta** and you get out θ, that has nothing to do with the value of the variable **theta**. θ is not the value of **theta**, it is just a different way of printing the variable name (at this point, **theta** is still an unassigned variable). You can use these names as assigned variables just like any other names.

```
> alpha := 2;
> beta := x^2-1;
```

And you can use these names as unassigned variables in expressions just like any other names.

```
> theta^2+cos(theta);
> (lambda^2 + eta*rho)/(omega + sigma*epsilon);
```

However, there are a few of Greek letters that are reserved names.

```
> gamma := 0; Beta := 0; Chi := 0; Pi := 0; Psi := 0; Zeta := 0;
```

Notice that **Pi** is not capital pi.

```
> Pi;
```

So how do you get the Greek letter capital pi? It is called **PI**.

```
> PI;
```

This is not a reserved name.

```
> PI := 100;
```

Also, Chi and Zeta do not represent capital chi and capital zeta.

```
> Chi, Zeta;
```

Here is how you get these last two capital letters.

```
> CHI, ZETA;
>
```

At the beginning of this section we said that a variable name could be "almost any string of characters". In fact this is not quite true. You can make *any* string of characters into a variable name (except for the keywords). But if you want to put "weird" characters in a name, you need to use left-quotes around the name. For example, the following variable name follows the rules for variable names that we gave above.

```
> variable_name := 1;
```

But the next variable name is not allowed since it has a space in it (in fact, it is really two names).

```
> variable name := 2;
```

But if we put left-quotes around the last attempt, then the space in the name is allowed.

```
> `variable name` := 2;
```

Notice that the left quotes are not really part of the name; they do not appear in the output. But whenever we want to refer to this name, we must refer to it using the left quotes.

```
> `variable name`;
```

The variable name in the next command is not valid because of the + character in the name.

```
> variable+name := 3;
```

But if we use left quotes again, then the + character is not a problem.

```
> `variable+name` := 3;
```

We just need to always use the left quotes when we refer to this variable.

```
> `variable+name` + `variable+name`;
```

Here is a very extreme case of a weird variable name. The name of this variable is **2+2**.

```
> `2+2` := 5;
```

Now we can write what seem to be nonsense formulas.

```
> evalb( `2+2`+`2+2`=10 );
> evalb( `2+2`^2=25 );
```

If we really insisted on it, we could make a number into a name.

```
> `2` := 5;
```

So now we have the following true equation.

```
> evalb( `2`= 5 );
```

The next (nearly unreadable) command uses right and left quotes to simultaneously unassign the names **`2+2`** and **`2`**.

```
> `2+2`, `2` := '`2+2`', '`2`';
> 
```

Exercise: How could you use Maple to misinterpret the equation $x^2 - 2\,x - 1 = 0$ and make **x^2-2*x-1** a name for zero?

```
> 
```

It is not a good idea to use left quoted variable names in your worksheets. They are too difficult to use and to read. So what are they good for? Mostly they are used internally by Maple. For example, Maple has a number of predefined functions that have names that are left quoted. All of these functions use a **/** character in their name and they have the name of some well know Maple

command before the `/`. These functions are "helper functions" that handle special cases for the well known Maple command. So for example, `` `simplify/trig` `` and `` `simplify/sqrt` `` are special cases of the **simplify** command, `` `convert/trig` `` and `` `convert/exp` `` are special cases of the **convert** command, `` `solve/linear` `` is a special case of the **solve** command, and `` `combine/trig` `` is a special case of the **combine** command. Here is an example of a Maple command that really uses a "helper function" (the helper function is specified by a second parameter to the **convert** command).

```
> convert( arctanh(x), ln );
```

Here is how we can call the helper function directly.

```
> `convert/ln`( arctanh(x) );
```

There are a lot of these specially named helper functions but we rarely need to use them ourselves since Maple uses them automatically whenever it needs to. But you will see a lot of references to them in Maple's online help. For example, since **combine/trig** is the special case of **combine** that handles trigonometric expressions, the online help for **combine** refers you to the online help for **combine/trig** for the details on how Maple simplifies trigonometric expressions. To see this, place the cursor on this **combine** and press the F1 key to call up the help page and then do the same with **combine/trig**.

```
>
```

Exercise: What happens if you have a variable name with a `/` in it but you do not left quote the name?

```
>
```

```
>
```

2.6. Indexed names

In the last section we said that Maple has two kinds of variable names, symbols and indexed names. Here we explain what an indexed name is.

It is very common in math books to see variables with subscripted indices like these (x_1, x_2, x_3). Subscripted variables can be used to represent the entries of a vector or a matrix, or the terms of a sequence. In Maple we use **indexed names** to represent these subscripted variables. We enter an indexed name into Maple using square brackets around the index.

```
> x[1], x[2], x[3];
```

Notice that x_1 is not the same as *x1*. The former is an indexed name and the later is just a regular variable name (a symbol) that happens to be made up of the characters x and 1.

```
> x[1], x1;
```

```
>
```

Pretty much anything we want can go inside the square brackets. Here are some unusual examples.

```
> x[hello_there];
```

```
> x[evalf(ln(2))];
```

Maple will do any simplification of the index that it can.

```
> x[1+3];
> x[i+i];
> i := 2; j := 3;
> x[i+j];
> a := b; b := c; c := d;
> z[a];
>
```

We can put more than one subscript on a name.

```
> x[a,b];
```

We can put a subscript on a subscript.

```
> x[a[b]];
```

We can also put a subscript on a subscripted name.

```
> x[a][b];
```

Notice the difference in the way the last two examples were entered into Maple. The first has a subscripted subscript and the second example is a subscripted name with a subscript. If you look carefully at the outputs, you should notice a slight difference in the way these two examples are typeset.

```
>
```

Exercise: What kind of names are the following two examples?

```
> x[a][y[b]];
> x[a[y]][b];
>
```

We can assign values to indexed names.

```
> x[1] := 0;
> x[2] := Pi;
> x[3] := 1/sqrt(2);
```

And we can recall these values.

```
> x[1], x[2], x[3];
>
```

An indexed name has two parts. There is the **header**, which is the name that is on the left of the brackets (and which may itself be an indexed name), and there is the **selection operation**, which is made up of the brackets and what is contained in them. The details of what we can do with an indexed name depends a lot on what the header of the indexed name points to. For example, the header of an indexed name can point to a list or an expression sequence, as in the following commands.

```
> l := [a, b, c];
> s := a, b, c;
```

We can use an indexed name as a name for each of the middle elements.

```
> l[2];
> s[2];
```

We can make an assignment to the indexed name `l[2]` because the header `l` evaluates to a list, but we cannot make an assignment to the indexed name `s[2]` because the header `s` points to an expression sequence.

```
> l[2] := 5;
> s[2] := 5;
```

As we just said, the rules for using an indexed name depend on what the header points to. We will go over various special cases of using indexed names in several different worksheets. In the remainder of this section we briefly discuss one important use of indexed names.

```
>
```

When we assign a value to an indexed name and the header is an unassigned name, we are implicitly telling Maple to create something called a **table** to hold the value, and the header of the indexed name becomes the name of the table. For example, the subscripted names `x[1]`, `x[2]`, and `x[3]` all have the same header `x`, so the three earlier assignment to these names all put values into the same table called `x`. We can look inside the table `x` with the following command.

```
> op( x );
```

The following two commands will also display the whole table for us.

```
> print( x );
> eval( x );
```

Notice that the following command tells us nothing about `x` (this is very similar to the way that names of Maple functions behave).

```
> x;
```

Here is another example of (implicitly) creating a table.

```
> fruit[apple] := yum;
> fruit[orange] := delicious;
> fruit[peach] := my_favorite;
```

Here are the contents of the table `fruit`.

```
> op( fruit );
>
```

You can also explicitly create a table (as opposed to implicitly creating one by assigning to an indexed name with an unassigned header). The **table** command explicitly creates a table.

```
> table( [sin(alpha), cos(alpha), tan(alpha)] );
```

But the last command created an "anonymous table". The table it created does not have a name yet. Let us give this table the name `T`.

```
> T := %;
```

Now we can use indexed names with the header `T` to refer to individual entries in the table.

```
> T[1], T[2], T[3];
```

The following command adds a new entry into the table `T`.

```
> T[5] := sec(alpha);
```

```
> op( T );
>
```

We can unassign an indexed name in two ways. We can unassign a specific indexed name.

```
> T[5] := 'T[5]';
```

Notice that the last command removed one entry from the table **T**.

```
> op( T );
```

Or we can unassign all of the indexed names associated with a table by unassigning the name of the table.

```
> T := 'T';
> op( T );
```

Now **T** is an unassigned variable and it no longer represents a table. And each of the indexed names **T[1]**, **T[2]**, and **T[3]** is also unassigned now.

```
> T[1], T[2], T[3];
>
```

There is a special case of a table called an **array** (in an array, all of the indices must be integers). And there are special cases of array called **vector** and **matrix**. We will say more about these object when we get to the worksheet about Maple's data structures.

```
>
```

2.7. Concatenated names

If we have two names, say **hot** and **dog**, combining the two names into one name, **hotdog**, is called **concatenation**. Maple has a special operator for concatenating names, the **concatenation operator**.

```
> hot||dog;
```

A name like **hot||dog** is sometimes called a concatenated name. Maples evaluates concatenated names into the concatenation of the names that are on either side of the vertical lines, so the vertical lines really are *not* part of the name. Actually, you do not need to have a name on both sides of the vertical lines, just on the left side. For example **x||1** does not have a name on both sides of the vertical lines, but it evaluates to the name **x1**.

```
> x||1;
```

On the other hand, **1||x** cannot be evaluated.

```
> 1||x;
```

Maple will try to evaluate what is on the right side of the concatenation operator. Suppose we make the following assignment.

```
> x := y;
```

Now see what Maple makes of **x||x**.

```
> x||x;
```

Notice that Maple evaluated the **x** on the right side of the operator but not the **x** on the left side. Now make another assignment.

```
> i := 2;
```

Try **x||i**.
```
> x||i;
```
Maple evaluated the **i** but not the **x**. You can even do some arithmetic on the right side of the operator.
```
> x||(i+2*i);
```
But you cannot get too fancy.
```
> x||(i/3);
```
Notice that the concatenation operator did not evaluate, so the result is not a name.
```
>
```


So what is the concatenation operator good for? It is used mostly for generating automatic sequences of names. Here is an example.
```
> seq( x||i, i=1..20 );
```
Here is another way that the concatenation operator can automatically generate a sequence of names. We can put a list of suffixes inside parentheses on the right hand side of the operator, and the operator will create a sequence of names.
```
> x||(1,2,3);
> help||(s, ed, ing, er);
```
We can specify a range of integers inside the parentheses and easily get a large number of names.
```
> x||(1..20);
```
The range can be determined by letters also, but we need to use a special notation.
```
> x||("c".."k");
```
(Look carefully at that last output.) We can even double up with the vertical lines.
```
> x||(1,2)||(a,b,c);
```
This trick does not work on the left hand side of the operator.
```
> (a,b,c)||x;
```
We can use this trick to perform multiple assignments with one command.
```
> x||(1,2,3) := apples, oranges, plums;
> x||(1..3);
> z||(1..10) := seq( i^2, i=1..10 );
> z||(1..10);
```
Now here is a puzzle. How do we change these last ten variables back into unassigned variables? The following command does not work.
```
> z||(1..10) := 'z||(1..10)';
```
We could change the variables back one at a time, but that would not be practical if there were 100 (or 1,000,000) variables. We could use the **restart** command, but that would make all of our variable unassigned and that might not be desirable. The following command does work. We will see in the worksheet on Maple's evaluation rules why it does.
```
> z||(1..10) := evaln( z||(1..10) );
> z||(1..10);
>
```


What are sequences of names like **x1** through **x20** good for? Here is a more detailed example.

Suppose we wanted to generate a sequence of random polynomials and give each polynomial in the sequence a unique name. The following command will generate one random polynomial in the variable **x**.

```
> randpoly( x );
```

The next command will automatically generate six random polynomials in **x** and it could easily be changed to generate any number of polynomials.

```
> seq( randpoly(x), i=1..6 );
```

The easiest way to come up with a unique name for each polynomial would be to use something like **p1**, **p2**, **p3**, etc., up to however many polynomials we want to generate. The concatenation operator was designed for generating a sequence of names just like this. The following command creates six random polynomials and uses the concatenation operator to create a unique name to assign to each polynomial.

```
> p||(1..6) := seq( randpoly(x), i=1..6 );
```

Now we can check that the names really do have values.

```
> p1; p2; p3; p4; p5; p6;
```

Here is another way to assign random polynomials to automatically generated names. The next command creates six equations, each one with a concatenated name on the left hand side and a random polynomial on the right hand side. Then the second command uses **assign** to cause the equations to be assignments.

```
> seq( q||i=randpoly(x), i=1..6 );
> assign( % );
```

Check that the names really have been assigned values.

```
> q1; q2; q3; q4; q5; q6;
```

Here is still another way to assign random polynomials to automatically generated names. (We will discuss Maple's for-loop in a later worksheet.)

```
> for i from 1 to 6 do r||i := randpoly(x) od;
>
```

Six is not all that high a number. The following command generates 1,000 random polynomials and assigns each of them a unique name. Notice the colon at the end of the command so that we do not see the whole list!

```
> p||(1..1000) := seq( randpoly(x), i=1..1000 ):
```

Now we can examine some of our random polynomials.

```
> p475;
> p803;
```

Generating *and labeling* large amounts of data like this is very common when using Maple, and it would be much more difficult to do without using concatenated names.

```
>
```

The 1,000 variable names **p1** through **p1000** that we generated above can all be unassigned by the following command (notice the colon at the end of the command).

```
> p||(1..1000) := evaln( p||(1..1000) ):
> p475, p803;
```

```
[ >
```

Alternatively, we can assign 1000 random polynomials to the names **p1** through **p1000** this way.

```
[ > for i from 1 to 1000 do p||i := randpoly(x) od:
[ > p803;
```
And then we can unassign these 1000 names like this.

```
[ > for i from 1 to 1000 do p||i := evaln(p||i) od:
[ > p803;
[ >
```

We can mix concatenated names with indexed names.

```
[ > seq( v[i||j], j=1..4 );
[ > for j from 1 to 4 do v[i||j] := x||j od;
[ > op( v );
[ >
```

We can do some strange things with the concatenation operator. For example:

```
[ > y||(-5..5);
[ > y||(-2) := Pi;
[ > y||(-2);
[ >
```

Exercise: How else might we refer to the variable **y||(-2)**?

```
[ >

[ >
```

2.8. Online help for variables and names

Maple quite often uses the terms variable, symbol, and name interchangeably, so the following three commands all call up the same help page.

```
[ > ?variable
[ > ?symbol
[ > ?name
```
The following command brings up the help page for indexed names.

```
[ > ?indexed
```
The brackets that are part of an indexed name are called the selection operation. The following help page describes this operation.

```
[ > ?selection
```
Alternative definitions of what a name, a symbol, and an indexed name are can be found in the next three help pages. These are help pages for Maple's data type command, which we will say much more about in the worksheet about Maple's data structures.

```
[ > ?type,name
[
```

`> ?type,symbol`

`> ?type,indexed`

Indexed names are used for tables and arrays. We will say more about these when we get to the worksheet about Maple's data structures.

`> ?table`

`> ?array`

Indexed names are also used for vectors and matrices, which are special cases of arrays.

`> ?vector`

`> ?matrix`

Maple makes a distinction between left quoted names and double quoted strings. We will not have much use for strings. The following command brings up the help page for them.

`> ?string`

The distinction between names and strings can be a bit confusing. For example, the following help page, which is about the "null string" is not really about strings at all. It should be called the "null name". (It seems that this help page never got updated after strings were introduced in Maple V Release 5).

`> ?nullstr`

The help pages for the concatenation operator and the con**cat**enation command for names (and strings) are called up by the next two commands.

`> ?||`

`> ?cat`

We mentioned that **I** is an example of an alias. Here is the help page that describes aliases.

`> ?alias;`

Using right quotes to convert an assigned name into an unassigned name is described in the next help page.

`> ?uneval`

The last help page also describes the special case of unassigning a name where, instead of using right quotes, you need to use the **evaln** command.

`> ?evaln`

We have tried to emphasize the distinction in Maple between the assignment operator **:=** and the equal sign **=**. The next help page describes the assignment operator.

`> ?assignment`

A brief description of the equal sign **=** (and the other relational operators) is contained in the next help page.

`> ?equation`

The **assign** command, which in certain cases lets an equation act like an assignment, is described in the next help page.

`[`

> **?assign**

We briefly mentioned making multiple assignments with a single assignment operator. This was introduced in Maple V Release 5.

> **?updates,R5,language**

We have said that Maple has a lot of reserved names that you should not (or cannot) use as your own variable names. The list of Maple keywords is brought up by the next command. These are not really names and you cannot use them as names.

> **?keyword**

Maple's list of initially known names is brought up by the next command. You could, if you wanted to, use these names for your own variables, but it is not a good idea to do so.

> **?ininame**

Maple's predefined constants, which are a subset of the initially known names, are listed in the following help page.

> **?constants**

Here are two different lists of Maple's initially know function names.

> **?inifcn**

> **?index,function**

Here is a list of all the names reserved for packages.

> **?index,package**

These list are not all of the reserved names in Maple. For example, **fraction** and **quadratic** are reserved names that do not appear in any of the above lists.

> **fraction := 0;**

> **quadratic := 0;**

I do not know of a complete list of Maple's reserved names.

If you should ever really insist on using a reserved name for one of your own variables, you will need to use the **unprotect** command.

> **?unprotect**

>