

Expressions vs. Functions vs. Programs in Maple

One issue that often causes confusion is the distinction Maple makes between "expressions" and "functions". Sometimes, the same confusion exists in ordinary mathematics, between "variables" and "functions". The idea is pretty straightforward, but can be hard to keep track of sometimes.

For example, suppose you want y to be x^2 . There are two senses (in mathematics and in Maple) in which this can be taken. First, you may simply want y to be a variable that represents the square of whatever x happens to be. Then in Maple, you would write:

```
> y:=x^2;
```

$$y := x^2$$

(In mathematics, you'd write pretty much the same thing, perhaps without the colon). This defines y to be the *expression* x^2 . The other thing you might mean is that y should represent a *function* that transforms any number (or variable) into its square. Then in Maple you would write:

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

You read this as " y maps x to x squared". In ordinary math, you would probably write $y(x)=x^2$, more or less.

In calculus classes, we are often casual (to the point of being careless) about the distinction between variables and functions. This is one reason the chain rule for derivatives can be so confusing. Maple, however, forces us to be explicit about whether we are using expressions or functions. Most of the time, we can accomplish what we want to do with either of them (although the way we have to do things can be somewhat different, as illustrated below), but occasionally we are forced to use one or the other.

We look at several (most, we hope) of the operations you will need to apply to functions and expressions -- sometimes, we will use commands that are described elsewhere in this manual without too much explanation; you can find more complete explanations in the sections dealing with these commands. The operations we will look at are "plugging a value (or other expression) in for x" into y, solving equations involving y, applying calculus operations (limit, derivative and integral) to y, and plotting the graph of y.

EXPRESSIONS

Even though it is not such an interesting expression, we will go through all of the operations on the expression x^2 . So all of the examples below assume that the statement

```
> y:=x^2;
```

$$y := x^2$$

has been executed first.

1. **Plugging in:** To plug a number (or other expression) in for x and evaluate y, the command subs (short for substitute) is used. The syntax is explained fully in the section on the subs command. Here is a simple example:

```
> subs(x=8,y);
```

$$64$$

It is possible to make substitutions of one variable for another as well:

```
> subs(x=f,y);
```

$$f^2$$

You can even substitute expressions involving x for x -- such as when you calculate derivatives by the definition (i.e., the long way):

```
> subs(x=x+h,y);
```

$$(x+h)^2$$

The important thing to remember about subs is that it has NO EFFECT on y at all. It

just reports what the result would be if you make the substitution. So even after all the statements above, the value of y is still:

```
> y;
```

$$x^2$$

2. Solving equations: One helpful use of expressions is that they save typing. You can use the name of an expression in its place anywhere you might need to type the expression. Not that x^2 is so onerous to type, but you can imagine more substantial uses. To solve quadratic equations (for instance $x^2=5$), we can type y instead of x^2 , as follows:

```
> solve(y=5);
```

$$\sqrt{5}, -\sqrt{5}$$

This works (of course) with `fsolve`, too:

```
> fsolve(y=5);
```

$$-2.236067978, 2.236067978$$

Here is a more sophisticated example (note that we have to tell Maple what variable we are solving for because there are more than one):

```
> solve(y=a^2+4*a+4,x);
```

$$-2 - a, 2 + a$$

3. Calculus operations: The syntax for calculus operations is just as described in their respective sections. For instance to calculate the limit of x^2 as x approaches 5, we would type:

```
> limit(y,x=5);
```

$$25$$

Or the derivative of x^2 :

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

$$2x$$

Or the indefinite integral:

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

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

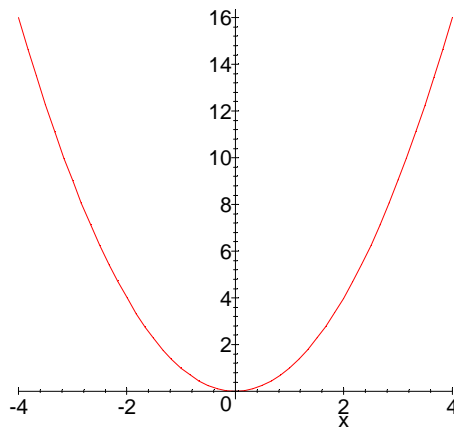
Or a definite one:

```
> int(y,x=-2..2);
```

$$\frac{16}{3}$$

4. **Plotting:** To plot the graph of y , the standard syntax of the plot statement is used:

```
> plot(y,x=-4..4);
```



An important aspect (which contrasts with using functions) is that the variable appears in the specification of the domain (i.e., the "x" in $x=-4..4$).

In general, expressions are easier to define and use than functions, as we shall now see:

FUNCTIONS

Again, we'll stick with defining y to be x^2 . But this time we're assuming that y has been defined as a function, via the statement:

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

$$y := x \rightarrow x^2$$

1. **Plugging in:** This is the one operation that is actually easier for functions. Since

the definition of y is now:

```
> y;
```

y

invisible to the usual way of looking at variables (because y is not a variable, it is a function). But you can look at the definition of y using the notation of standard mathematics:

```
> y(x);
```

x^2

Since y is a function, we can use other letters or expressions in place of x in y(x):

```
> y(t);
```

t^2

```
> y(5);
```

25

```
> y(x+h);
```

$(x+h)^2$

You get the idea.

2. Solving equations: To solve an equation involving a function, you need to plug in a variable in order to convert it to an expression, because equations have expressions on either side of the equals sign. (Note that y is a function, but y(x) is an expression):

```
> solve(y=2,x);
```

This gives no output because y is a function, not an expression. We should type instead:

```
> solve(y(x)=2,x);
```

$\sqrt{2}, -\sqrt{2}$

It is the same with fsolve :

```
> fsolve(y(x)=17,x=3..5);
```

4.123105626

(See the fsolve section of this manual for the rest of the syntax here).

3. **Calculus:** Maple actually has a special command for taking derivatives of functions (but none for limits or integrals). It is called D. You don't need (in fact, you shouldn't use) the $y(x)$ notation with D, just y :

```
> D(y);
```

$$x \rightarrow 2x$$

Notice that the result of D is another function (here, the function that maps x to $2x$) rather than an expression, as is the case with `diff`.

For integrals and limits, you must use the $y(x)$ notation as with equations. It is also possible to use `diff` with the $y(x)$ notation. Here are some examples:

First, one that doesn't work:

```
> limit(y,x=3);
```

$$y$$

(See why we need the $y(x)$ notation?)

```
> limit(y(x),x=3);
```

$$9$$

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

$$2x$$

Of course, one advantage of functions is that we can also do:

```
> diff(y(t),t);
```

$$2t$$

if we need to. Finally, for integrals:

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

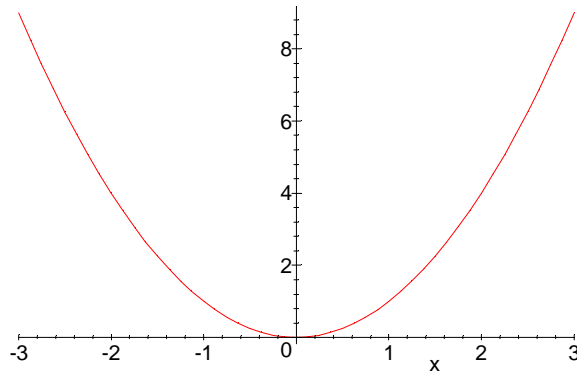
$$\frac{1}{3}x^3$$

```
> int(y(q),q=1..4);
```

$$21$$

4. **Plotting:** There are two ways to do plotting of functions. The first is the usual way as for expressions, and works the same as the calculus rules with $y(x)$:

```
> plot(y(x),x=-3..3);
```

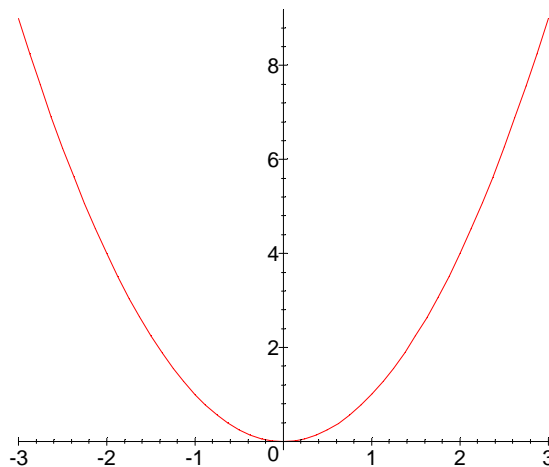


No surprises here -- all of the usual plotting options and tricks are available for functions this way.

One new wrinkle is that a function can be plotted without using the $y(x)$ notation, but the syntax is slightly different: Since no variable is specified in the notation, no variable can be specified for the domain (otherwise an "empty plot" will result).

Here is the proper syntax:

```
> plot(y,-3..3);
```



Compare this graph with the one above -- there is one subtle difference (the x-axis isn't labelled).

That's basically it, as far as functions and expressions are concerned. The idea of

using functions opens up a major new part of the Maple system -- the fact that in addition to being a "calculus calculator", it is also a *programming language*. Yes, you can write programs in Maple. This will occasionally be useful, because occasionally you will need to apply some procedure (like taking the derivative and setting it equal to zero) over and over, and you will get tired of typing the same thing all the time. You can create your own extensions to the Maple language in this way.

Our purpose here is not to teach Maple programming, but here are a few basics and an example or two so you see how it's done. There are several examples of Maple programs in the demonstrations. You can copy and paste them in your homework assignments if you find this useful (you will, occasionally).

A Maple program (actually called a "procedure") is created by a Maple statement containing the `proc` command. A simple example is the following:

```
> y:=proc(x) x^2 end;
```

```
y := proc(x) x^2 end
```

This statement is in fact equivalent to `y:=x->x^2`. In fact, Maple translates `y:=x->x^2` into the above statement. It illustrates most of the basic parts of a Maple program:

1. `y:=` -- it is necessary to give your program a name. You do this by assigning the procedure to a variable name.
2. `proc` -- This tells Maple that you are writing a program.
3. `(x)` -- You put the names of the input to the program in parentheses. In this case, the program takes one piece of input, the number `x`.
4. `x^2` -- This is the statement that will be executed when the program is run. There can be many statements in a procedure. They are separated by semicolons, as usual.

5. `end;` -- All programs must end with an `end;` statement, otherwise, how is Maple to know you're done?

Once the program is defined, you can use it just as you would any Maple command:

```
> y(20);
```

400

and so forth...

A more interesting program is the following one, which looks for critical points (it doesn't always work, though).

```
> crit:=proc(f) local d;  
    d:=diff(f,x);  
    print(solve(d=0,x),` is a critical point for `,f);  
end;
```

```
crit := proc(f) local d; d := diff(f,x); print(solve(d=0,x),` is a critical point for `,f) end
```

Notice that Maple parrots back the definition of the function. This function needed two statements, and has a "local" variable. To get more than one line in your definition of a function, you need to use the SHIFT+ENTER keys together between lines, and just plain ENTER at the end.

The "local" variable means that "d" is declared to be a "private" variable for the purposes of the program -- so the use of d here will NOT interfere with any other definition of d that may be hanging around outside of the program. Local variables must be declared right at the beginning of the program, and a semi-colon must follow the list of local variables.

The other new thing here is the "print" statement. When we use the program "crit", you will be able to figure out what the print statement does:

```
> crit(x^2+2*x);
```

-1, *is a critical point for , $x^2 + 2x$*

So the value of d (calculated in the first statement of the program) is printed, followed by the character string ``is a critical point for`` (note that both of the quotes that surround a character string are LEFT quotes -- on the keyboard to the left of the digit 1), and then the input expression is printed.

A TINY BIT OF PROGRAMMING: We can improve `crit` so that it checks to see if the critical point is a local max or a local min using the second derivative test, as follows:

```
> crit:=proc(y) local d, dd, c;  
    d:=diff(y,x);  
    c:=solve(d=0,x);  
    print(c, ` is a critical point for `,y);  
    dd:=diff(d,x);  
    if subs(x=c,dd)>0 then print(`It is a local minimum.`)  
    elif subs(x=c,dd)<0 then print(`It is a local maximum.`)  
    else  
    print(`The second derivative test is inconclusive for this  
point.`) fi;  
end;
```

```
crit := proc(y)  
local d, dd, c;  
    d := diff(y, x);  
    c := solve(d = 0, x);  
    print(c, ` is a critical point for `, y);  
    dd := diff(d, x);  
    if  $0 < \text{subs}(x = c, dd)$  then print(`It is a local minimum.`)  
    elif  $\text{subs}(x = c, dd) < 0$  then print(`It is a local maximum.`)  
    else print(`The second derivative test is inconclusive for this point.`)  
    fi  
end
```

The fancy thing here is the "if" statement. In fact, it is an "if/then/elif/then/else/fi" statement! You read "elif" as "else, if". The

statement embodies the second derivative test: If the second derivative dd at the point c is positive then c is a local minimum, otherwise, if dd is positive at c , then c is a local maximum, otherwise the test is inconclusive. The "fi" at the end tells Maple that the if statement is over (it is possible to have several statements after each "then" and after the "else". "fi" is "if" backwards, sort of like a right parenthesis is to a left one.

Here is the program at work:

```
> crit(x^2+2*x);
```

*-1, is a critical point for , $x^2 + 2x$
It is a local minimum.*

```
> crit(x^3+2*x);
```

$\frac{1}{3}I\sqrt{6}, -\frac{1}{3}I\sqrt{6}$, is a critical point for , $x^3 + 2x$

Error, (in crit) invalid types in sum

Uh-oh-- We told you it wouldn't always work! This is because x^3+2x has two (complex!) critical points and our program hasn't made allowances for either the fact that the critical points might not be real nor that there might be more than one.

Here is a complete, working version (it still doesn't check for errors in the input, but it's not so bad). You can look in books about Maple for all of the syntax, or find some of it in Maple help:

```
> crit:=proc(y) local d, dd, c, cc;
  d:=diff(y,x);
  c:=solve(d=0,x);
  for cc in c do if type(cc,realcons) then
    print(cc,` is a critical point for `,y);
    dd:=diff(d,x);
    if evalf(subs(x=cc,dd))>0 then print(`It is a local minimum.`)
      elif evalf(subs(x=cc,dd))<0 then print(`It is a local maximum.`)
    else
      print(`The second derivative test is inconclusive for this
```

```

point.`) fi;
fi; od;
end;
crit := proc(y)
local d, dd, c, cc;
d := diff(y, x);
c := solve(d=0, x);
for cc in c do
if type(cc, realcons) then
print(cc, ` is a critical point for `, y);
dd := diff(d, x);
if 0 < evalf(subs(x=cc, dd)) then print(`It is a local minimum.`)
elif evalf(subs(x=cc, dd)) < 0 then print(`It is a local maximum.`)
else print(`The second derivative test is inconclusive for this point.`)
fi
fi
od
end

```

It is nice that when Maple parrots the definition of the program, it formats the output so you can see where the various `if`'s and `do`'s begin and end. Let's try it out:

First on one that worked before:

```
> crit(x^2+2*x);
```

-1, *is a critical point for*, $x^2 + 2x$
It is a local minimum.

Now a better one:

```
> crit(x^3+2*x);
```

No output! That is right, for as we know from before, this function has no real critical points. If we change it a little, it will have two:

```
> crit(x^3-2*x);
```

$\frac{1}{3}\sqrt{6}$, *is a critical point for*, $x^3 - 2x$
It is a local minimum.

$-\frac{1}{3}\sqrt{6}$, is a critical point for $x^3 - 2x$

It is a local maximum.

Just as it should be!

One place where you will need to write a program like this is to define "piecewise" functions. Here is an example, from which you will get the idea:

```
> f:=proc(x)
  if x<2 then x-5 else 3-2*x^2 fi
end;
```

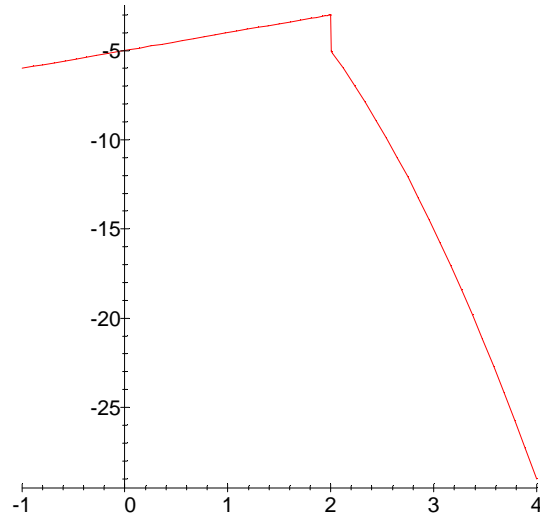
$f := \text{proc}(x) \text{ if } x < 2 \text{ then } x - 5 \text{ else } 3 - 2 * x^2 \text{ fi end}$

You have to be careful when plotting these. For instance the following won't work:

```
> plot(f(x),x=-1..4);
Error, (in f) cannot evaluate boolean
```

However, as indicated in the section of this manual on the `plot` function, the following does work:

```
> plot(f,-1..4);
```



The other programming construction we will use occasionally (even outside of programs) is the "for/from/to/do/od " statement. It is used to make tables of functions and for repetition of a set of commands over and over. For example, a good table of sines and cosines obtained from the following:

```
> for k from 1 to 8 do k*Pi/4,cos(k*Pi/4),sin(k*Pi/4) od;
```

$$\frac{1}{4}\pi, \frac{1}{2}\sqrt{2}, \frac{1}{2}\sqrt{2}$$

$$\frac{1}{2}\pi, 0, 1$$

$$\frac{3}{4}\pi, -\frac{1}{2}\sqrt{2}, \frac{1}{2}\sqrt{2}$$

$$\pi, -1, 0$$

$$\frac{5}{4}\pi, -\frac{1}{2}\sqrt{2}, -\frac{1}{2}\sqrt{2}$$

$$\frac{3}{2}\pi, 0, -1$$

$$\frac{7}{4}\pi, \frac{1}{2}\sqrt{2}, -\frac{1}{2}\sqrt{2}$$

$$2\pi, 1, 0$$

You get the idea -- the statement $k \cdot \pi/4, \cos(k \cdot \pi/4), \sin(k \cdot \pi/4)$ was executed for each value of k from 1 to 8. The "do" and "od" are used to surround what is allowed to be a set of several statements.