

## 5. Assignment and Unassignment

In this lecture we see the effects of the assignment (`:=`) and how to undo the assignment.

### 5.1 Verifying symbolic formulas

As example we start with solving a general cubic polynomial, where the coefficients are left as parameters:

```
[> p := x^3 + a*x^2 + b*x + c; # a general cubic polynomial
[> sols := solve(p,x); # have to tell Maple what we are solving for
```

Let us verify these formulas for a general choice of the parameters for which we know the roots. How could we do this? Well, we can choose three random numbers to be the roots of a polynomial. If we know the roots of a polynomial, we know its coefficients. We assign the known coefficients to the general ones and see what the formulas give us.

```
[> x1 := rand(); x2 := rand(); x3 := rand();
[> sp := (x-x1)*(x-x2)*(x-x3); # our special polynomial
[> spe := expand(sp); # we expand it to compute the coefficients
[> a1 := coeff(spe,x,2); b1 := coeff(spe,x,1); c1 := coeff(spe,x,0);
[> a := a1; b := b1; c := c1;
[> p; # this assignment has changed p
[> sols; # sols is sequence of three solutions
[> simplify(sols[1]); # try to simplify first solution
```

Here we run into a limitation of the symbolic simplifier. Let us turn to numerical calculations for rescue.

```
[> evalf(sols[1]); evalf(sols[2]); evalf(sols[3]);
[> x1;x2;x3; # compare with the chosen roots
```

With some goodwill, we recognize something of the chosen roots in the ones predicted by the exact formulas. To increase our faith in the computations, let us increase the precision:

```
[> evalf(sols[1],30); evalf(sols[2],30); evalf(sols[3],30);
```

We see that the imaginary parts of the solutions predicted by the formulas shrink and we recognize all digits in the chosen roots. With the specialization of the coefficients, we lost our general polynomial `p`:

```
[> p;
[> a := 'a'; b := 'b'; c := 'c'; # unassign with right quotes
[> p; # check if we have the general polynomial back
```

The above commands illustrate the use of right quotes to unassign a variable. Another use of the right quotes will be given in the next subsection.

### 5.2 Sharing and the side effects of the assignment

Sometimes we want variables to share the same value, and sometimes we don't. When a variable appears at the right hand side of an assignment, it is evaluated:

```
[> x := 10; y := x;
[> x := 11; y;
```

We see that, as `x` was evaluated to 10, `y` is independent of `x`. Suppose we want `z` to be just another name for `x`, then we can use the right quotes (as in the unassignment):

```
[> z := 'x'; z; # right quotes prevent evaluation
[> x := 12; y; z;
```

As `x` changes, also `z` changes, so `z` acts as a pointer to `x`.

### 5.3 The commands `assign`, `unassign`, and `evaln`

The `assign` and `unassign` commands have broader use than the constructions we have seen before. One example of the `assign` is to give formal parameters a solution value:

```
[> eqs := {y1+y2 = p1, y1 + p1*y2 = p2}; # linear system with parameters
[> vars := {y1,y2};
[> sols := solve(eqs,vars);           # solve for y1 and y2
```

Note that again we have to be cautious when using these formulas. But let us move on :

```
[> assign(sols);                       # turns the "=" into ":="
[> y1; y2;
[> unassign('y1','y2');               # observe the right quotes!
[> y1;y2;
```

The `assign` and `unassign` come in handy when the number of variables is huge, or not a priori known when used in Maple loops. The third possibility to undo an assignment is to work with `evaln` (evaluate to a name):

```
[> assign(sols);
[> y1 := evaln(y1); y2 := evaln(y2);
[> y1; y2;
```

### 5.4 Assignments

1. Consider the following sequence of commands:

```
[> restart;
[> x := y;
[> y := x + 1;
```

Maple will complain about this recursive assignment. How can you modify the last statement to avoid the error? Explain the solution.

2. Execute the following commands:

```
[> restart;
[> v[1] := 1: v[2] := 2: v[3] := 3:
[> for i from 1 to 3 do v[i] := 'v[i]'; end do;
[> v[1]; v[2]; v[3];
```

Explain the outcome of these commands.

Why does this loop not unassign the variables `v[1]`, `v[2]`, and `v[3]`?

Find one alternative *loop* to unassign the variables `v[1]`, `v[2]`, and `v[3]`.