

# 5. Writing Modules

---



## 5.1 Introduction

Objectives

## 5.2 The Need for Modules

The problem of writing functions with several steps

The solution

## 5.3 A Template for Using Module

The structure of a module

Example of a module

## 5.4 How to Develop a Module

Seven steps

Example problem using a module

Exercises

## 5.5 Advanced Concept: How Module Keeps Values Local

## 5.6 Problems

Problem 1: Manipulating data

Problem 2: Creating plots

Problem 3: Playing sound

## 5.1 Introduction

---

After the ability to write function definitions, the ability to write a **Module** is the next critical technique you need to learn. We have seen how the left hand side of a function summarizes what can possibly be a long and complex right hand side. Modules allow you to build your complex right hand sides in easy steps.

### Objectives

---

To understand the concept of module.

To understand when a module should be used in the design of a function.

To be able to write a function using a module.

To understand the steps in the development of a module.

## 5.2 The Need for Modules

---

### The problem of writing functions with several steps

---

Suppose you want to write a function where the right-hand side is most conveniently built up in two steps. For example suppose you wanted to write a function `ff` to calculate  $t^3 + 5t^2 + 7t + 6$ , but where  $t = x^4 + 2x^2 + 3x + 5$ .

You could define the function by substituting for `t` and writing the full right hand side in your function definition.

```
Clear[ff]
```

```
ff[x_] := 6 + 7 (5 + 3 x + 2 x^2 + x^4) +
          5 (5 + 3 x + 2 x^2 + x^4)^2 + (5 + 3 x + 2 x^2 + x^4)^3
```

However, this is not a *computationally efficient* solution to the problem, since  $x^4 + 2x^2 + 3x + 5$  is calculated *three* times when, for example if you were doing it by hand, you would only calculate it once each time the function needed to be evaluated.

Nor is it an *algorithmically elegant* solution, since the natural dependence of the expression on the parameter `t` has begun to be less obvious, especially to someone who did not design the function in the first place.

You can imagine also, that in many situations you may wish to design a function which has many more than just two steps in it. As the number of steps increases, a substitution like this would create an expression on the right hand side which was unacceptably unwieldy, unreadable and inefficient.

At first thought you might try to solve this problem by writing your function as a compound expression

```
Clear[ff]
```

```
ff[x_] := (t = x^4 + 2 x^2 + 3 x + 5; t^3 + 5 t^2 + 7 t + 6)
```

The function works correctly as we can verify, and only computes `t` once:

```
ff[x]
```

```
6 + 7 (5 + 3 x + 2 x^2 + x^4) +
5 (5 + 3 x + 2 x^2 + x^4)^2 + (5 + 3 x + 2 x^2 + x^4)^3
```

But it assigns a new value to **t** every time it is used:

```
{ff[2], t, ff[3], t, ff[4], t}
{49251, 35, 1507539, 113, 28839891, 305}
```

*This is not a good idea.* Our function definition should stand on its own completely, and not influence the definition of any other symbols. You can imagine what would happen if a function definition like the one above was embedded in a program you gave someone else to use, but they had been using the symbol **t** to represent something else of their own. Significant confusion!

This is an *extremely important* programming principle.

Do not generate loose-cannon symbols with your programs!

## The solution

Mathematica has solved this problem with the **Module** construct. The **Module** keeps the definition of any intermediate symbols *local*, that is, *internal* to the function definition so that their values outside the module remain unchanged. Modules are so useful, you will end up using them in a great many of your function definitions.

Rewriting the above function using a module gives us

```
Clear[ff]

ff[x_] :=
Module[{t}, t = x4 + 2 x2 + 3 x + 5; t3 + 5 t2 + 7 t + 6]
```

Note four things about a module:

- 1) The entire right hand side is enclosed within the module.
- 2) The first argument of the module is a list of symbols we want to use locally within the module only.
- 3) The second argument of the module is a compound expression.
- 4) The output of the module is the last expression in the compound expression.

With this definition, the value of **t** is not affected. For example, if we set **t** equal to **2**, then calculate with the function, the value of **t** is not affected (as it was in our function definition using a compound expression only).

```
t = 2; {ff[2], t, ff[3], t, ff[4], t}
{49251, 2, 1507539, 2, 28839891, 2}
```

## 5.3 A Template for Using `Module`

---

### The structure of a module

---

Modules are most useful when your program allows you to build up to your required output by a number of steps, each one using results of your previous steps. In the template below, your first computed expression **t1** involves just the function arguments. The next computed expression **t2** involves the function arguments and possibly, **t1**. The next involves the function arguments and possibly **t1** and/or **t2**. And so on.

The final computed expression is the output you want.

```

Clear[f]

f[x_,y_,z_,...] :=
Module[{t1,t2,t3,...},
t1 = expression involving x,y,z,... ;
t2 = expression involving x,y,z,...,t1 ;
t3 = expression involving x,y,z,...,t1,t2 ;
.....;
final output expression required]

```

Note carefully, that there should be no symbols used on the right hand side except

- 1) The function arguments **x, y, z, ...**
- 2) The symbols representing the intermediate (local) symbols **t1, t2, t3, ...** which must *all* be listed in the first argument of the module.
- 3) Inbuilt function symbols (like **Log, Sin, Integrate, ...**)
- 4) Symbols for functions of your own that you have defined and documented elsewhere.

As you can see from the template above, a program written as a module is straightforward to document and debug since the steps are generally sequential, and therefore each can be documented or debugged in turn.

### Example of a module

---

Suppose we wish to design a function which calculates a list of the roots of a quadratic equation  $\mathbf{a x^2 + b x + c}$ . The first decision we have to make is how to *represent* the quadratic equation as an argument to the function. We will discuss this important issue further as we proceed through the subject, but for the moment we take the simple approach of representing the equation  $\mathbf{a x^2 + b x + c}$  by the list of its coefficients **{a, b, c}**.

We know the solutions are given by the formula  $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ , so we *could* design the function as

```
Clear[quadraticRoots]

quadraticRoots[{a_, b_, c_}] :=
  {  $\frac{-b - \sqrt{b^2 - 4ac}}{2a}$ ,  $\frac{-b + \sqrt{b^2 - 4ac}}{2a}$  }
```

But this design is inefficient since we are calculating  $\sqrt{b^2 - 4ac}$  twice. A better design is to use **Module**:

```
Clear[quadraticRoots]

quadraticRoots[{a_, b_, c_}] :=
  Module[{d}, d =  $\sqrt{b^2 - 4ac}$ ; {  $\frac{-b - d}{2a}$ ,  $\frac{-b + d}{2a}$  }]
```

Testing gives us what we expect:

```
{quadraticRoots[{u, v, w}], quadraticRoots[{1, 2, 3}]}
```

$$\left\{ \left\{ \frac{-v - \sqrt{v^2 - 4uw}}{2u}, \frac{-v + \sqrt{v^2 - 4uw}}{2u} \right\}, \left\{ \frac{1}{2}(-2 - 2i\sqrt{2}), \frac{1}{2}(-2 + 2i\sqrt{2}) \right\} \right\}$$

## 5.4 How to Develop a Module

### Seven steps

Here are a few practical hints on the process of developing a program using a module. The steps are as follows

*Step 1:* Construct a typical example input expression and enter it.

*Step 2:* Perform successive operations on this example input (and perhaps other results generated along the way) leading to the output you want. Although these operations are being applied to a particular example, make sure you write each one so that it does not depend on the specific *detail* of the example you chose (so that it will work later with any other example). Assign a new name to the expression generated by each new operation.

*Step 3:* Copy and paste these successive operations into the compound statement of a module, putting a semi-colon after each one except the last, and making sure the sequence is the same one you developed them in.

*Step 4:* List any symbols from the compound statement which you want to remain local to the module in the symbol list at the beginning of the module. Symbols used as arguments on the left-hand side do not need to be listed here.

*Step 5:* Enter the function you have developed and test it out using the original example input you chose. Make sure it gives you the same result as you got when you performed each operation separately.

*Step 6:* Test out your function using various inputs. Pay particular attention to inputs that are special, but valid, cases. For example, most list processing programs should be able to deal with empty lists. These special cases are often what "breaks" a program.

*Step 7:* Document your program so that you or others can check, debug, or modify it at a later date. Head up your working with a Subsection title like "Implementation of ...". Before each operation write a short piece of text explaining what the operation is going to do.

## Example problem using a module

In this section we show the final result, then go through how we might have developed the solution to a simple programming problem using a module.

**The problem:** Construct a function called **plotFour** using **Module** that takes a plot already generated and constructs a plot consisting of the overlay of all the four plots resulting from successive rotations by  $90^\circ$ .

**The solution:** The solution, its implementation notes and its testing is shown below.

### ■ The plotFour function

*Usage:* **plotFour**[**p**] takes the output **p** of a function generated using **Plot** and constructs a plot consisting of the overlay of all the four plots resulting from successive rotations by  $90^\circ$ .

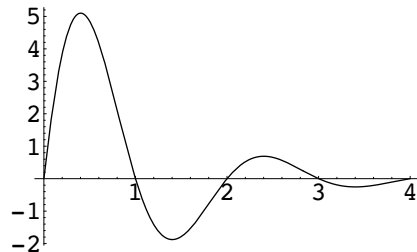
```
Clear [plotFour]

plotFour [p_] := Module[{R, p1, p2, p3},
  R[q_] := Show[q /. {x_, y_} => {-y, x}]; p1 = R[p];
  p2 = R[p1]; p3 = R[p2]; Show[p, p1, p2, p3]]
```

## ■ Implementation of plotFour

◆ *Step 1: Construct a typical example input expression and enter it.*

```
p = Plot[8 Sin[π t] Exp[-t], {t, 0, 4}]
```



- Graphics -

(You can make the plot smaller by clicking it and dragging a handle).

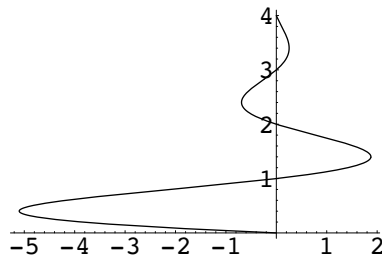
◆ *Step 2: Perform successive operations on this example input.*

Define an operation which rotates a graph anticlockwise by 90°

```
R[q_] := Show[q /. {x_, y_} -> {-y, x}]
```

Rotate the first plot by applying **R** to **p**

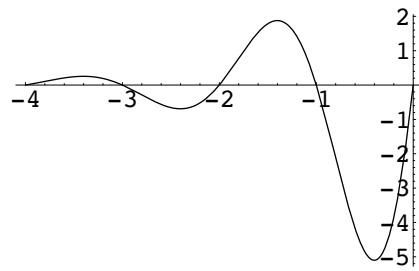
```
p1 = R[p]
```



- Graphics -

Rotate this second plot by applying **R** to **p1**

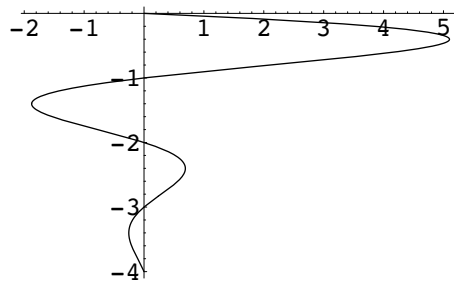
**p2 = R[p1]**



- Graphics -

Rotate this third plot by applying **R** to **p2**

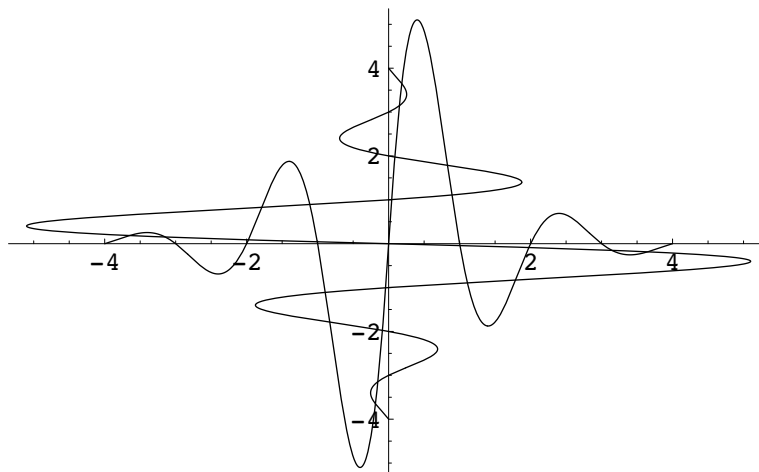
**p3 = R[p2]**



- Graphics -

Show all the plots together

**Show[p, p1, p2, p3]**

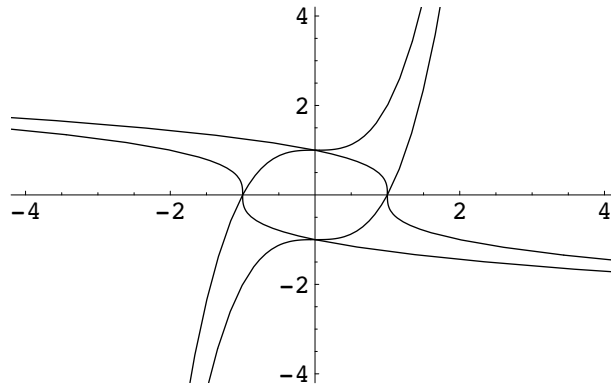


- Graphics -



◆ **Step 6: Test out your function using various inputs.**

```
plotFour[Plot[x3 + 1, {x, -2, 2}]]
```



- Graphics -

(You would generally do more extensive testing than just this one case.)

◆ **Step 7: Document your function**

See Appendix A: Packaging Your Program for some suggestions on ways to efficiently document your programs.

When you write a program, you are generally so totally involved in what you are doing that you believe you will always remember why you are doing it the way you are. This is a classic misconception! Unless the code is really simple, you will find your own code often quite inscrutable in as little as 6 months time. The objective of documentation is to give you and anyone else at least some clues to make this task easier.

## Exercises

---

◆ **Exercise: Writing a Module for data**

Write a function using **Module** and without using a replacement rule, which takes a list of data pairs, and creates a list of the first in each pair raised to the power of the second in that pair. That is,  $\{\dots, \{a, b\}, \dots\}$  becomes  $\{\dots, a^b, \dots\}$ .

◆ **Exercise: Writing a Module for matrix manipulation**

Write a function using **Module** which takes a matrix **A** and returns the pair of matrices  $\{A + A^T, A - A^T\}$ , where **A<sup>T</sup>** is the **Transpose** of **A**.



## 5.5 Advanced Concept: How Module Keeps Values Local

---

Module uses a mechanism similar to **Unique** to prevent confusion between the local temporary values of a symbol, **t** say, and its value assigned outside the module. Inside the module, it is not really **t** that is used, but another unique symbol based on the symbol **t**. We can see this by asking **Module** to return to us as output, a symbol that we have designated as local (by listing it in the first argument to the module).

```
{Unique[t], Module[{t}, t], Unique[t],
  Module[{t}, t], Unique[t], Module[{t}, t]}
{t$13, t$14, t$15, t$16, t$17, t$18}
```

## 5.6 Problems

---

### Problem 1: Manipulating data

---

Write a function called **AB** using **Module** which takes the pair **{A, B}** of lists of data, truncates the longer one to be the length of the shorter one, and produces a list whose elements are the products of the corresponding values in **A** and **B**.

*Hints:* You may find the functions **Length**, **Min**, and **Take** useful. Also, see what happens when you use ordinary multiplication with two lists of equal length.

Test your function on the following list pairs

```
{{}, {}}
{{a}, {}}
{{a}, {b}}
{{a, b}, {c}}
{{a, b}, {c, d}}
{{a, b}, {c, d, e}}
```



---

## Problem 2: Creating plots

---

Write a function called **dBPlotter** that takes a list of {frequency, sound pressure} pairs, converts the sound pressure to dB, cuts out the sound over a range of frequencies, and plots the result.

*Hint:* See the problems on manipulating sound data in the previous modules.



---

## Problem 3: Playing sound

---

Write a function called **dBPlayer** that takes a list of {frequency, sound pressure} pairs, and plays the result.

*Hint:* Check out the **Play** function and review some of the list processing functions you have already met. See if the listability property help. Review the exercises above.

*Hint:* The amplitude argument to the **Play** function could be any sum of terms of the form  $\mathbf{P}_i \mathbf{Sin}[2 \pi \mathbf{f}_i \mathbf{t}]$ , where  $\mathbf{P}_i$  is the pressure amplitude in Pa, and  $\mathbf{f}_i$  is the corresponding frequency in Hz.

