Lecture 3: Introduction to Mathematica II

Sept. 10 2007

3.016

Functions and Rules

Besides MATHEMATICA® 's large set of built-in mathematical and graphics functions, the most powerful aspects of MATHEMATICA® are its ability to recognize and replace patterns and to build functions based on patterns. Learning to *program* in MATHEMATICA® is very useful and to learn to program, the basic programmatic elements must be acquired.

The following are common to almost any programming language:

Variable Storage A mechanism to define variables, and subsequently read and write them from memory.

Loops Program structures that iterate. A well-formulated loop will always be guaranteed to exit[2].

Variable Scope When a variable is defined, what other parts of the program (or other programs) will be able to read its value or change it? The scope of a variable is, roughly speaking, the extent to which it is available.

Switches These are commands with outcomes that depend on a quality of variable, but it is unknown, when the program is written, what the variable's value will be. Common names are If, Which, Switch, IfThenElse and so on.

Functions Reusable sets of commands that are stored away for future use.

All of the above are, of course, available in MATHEMATICA® .

The following are common to Symbolic and Pattern languages, like MATHEMATICA® .

Patterns This is a way of identifying common structures and make them available for subsequent computation.

Recursion This is a method to define function that obtains its value by calling itself. An example is the Fibonacci number $F_n \equiv F_{n-1} + F_{n-2}$ (The value of F is equal to the sum of the two values that preceded it.) F_n cannot be calculated until earlier values have been calculated. So, a function for Fibonacci must call itself recursively. It stops when it reaches the end condition $F_1 = F_2 = 1$.

[2]Here is a joke: "Did you hear about the computer scientist who got stuck in the shower?" "Her shampoo bottle's directions said, 'wet hair, apply shampoo, rinse, repeat'."

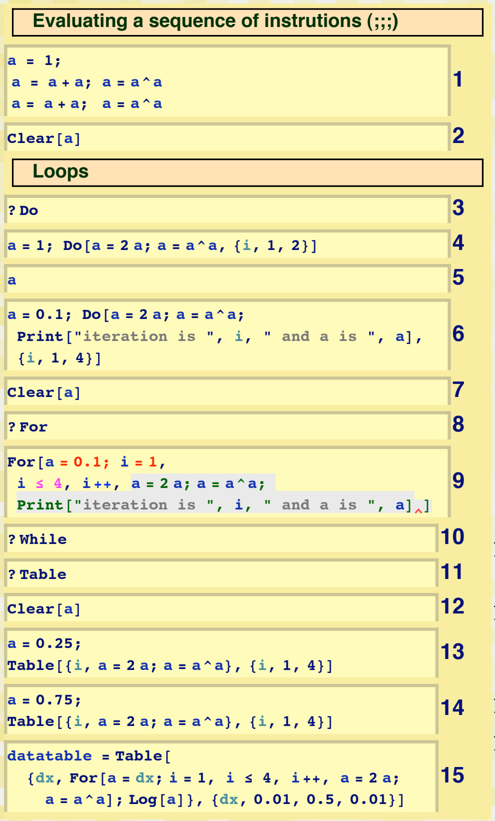
Lecture 03 MATHEMATICA® Example 1

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

3.016

Procedural Programming

Simple programs can be developed by sequences of variable assignment.



Evaluating a sequence of instrutions (;;;)

```
a = 1;
 a = a + a; a = a^a
 a = a + a; a = a^a                                        1

Clear[a]                                                   2
```

Loops

```
? Do                                                       3

a = 1; Do[a = 2 a; a = a^a, {i, 1, 2}]                     4

a                                                          5

a = 0.1; Do[a = 2 a; a = a^a;
  Print["iteration is ", i, " and a is ", a],              6
  {i, 1, 4}]

Clear[a]                                                   7

? For                                                      8

For[a = 0.1; i = 1,
  i ≤ 4, i++, a = 2 a; a = a^a;                            9
  Print["iteration is ", i, " and a is ", a]]

? While                                                    10

? Table                                                    11

Clear[a]                                                   12

a = 0.25;
Table[{i, a = 2 a; a = a^a}, {i, 1, 4}]                    13

a = 0.75;
Table[{i, a = 2 a; a = a^a}, {i, 1, 4}]                    14

datatable = Table[
  {dx, For[a = dx; i = 1, i ≤ 4, i++, a = 2 a;             15
    a = a^a]; Log[a]}, {dx, 0.01, 0.5, 0.01}]
```

1: Here is a simple program that is just a sequence of statements that reassigns `a` from an initial value (`a=1`). The program does this: take `a` add it to itself and assign the result back to `a`; raise this new `a` to the power `a` and assign back to itself. Repeat. In MATHEMATICA® , a semicolon— `;`—just indicates that output should be suppressed. There are five executions—two of them produce output on the screen.

3: However, it would be cumbersome and unaesthetic if we wanted to generalize the last two lines to many executions of the same type. This is where *program loops* come in. `Do` is a simple way to loop over an expression a fixed number of times. This is equivalent to item **1**, but could be easily generalized to more iterations.

4: The `Do` loop does not produce intermediate output, the current value of `a` can be obtained by asking MATHEMATICA® for the current value.

5: Here an equivalent example, but extra `Print` statements are added so that *intermediate output* can be observed.

8: A `For` loop is another loop structure that enforces good programming style: Its arguments provide: an initialization, an exit condition, an iteration operator, and a function statement, and is equivalent to item **6**, but it includes (a different) initial value for `a` in the `For` statement and iterates 4 times instead of 2.

9: The are many types of loop constructs; `While` is yet another.

10: `Table` is a very useful MATHEMATICA® iterating function. While it iterates, it leaves intermediate results in a List structure. Thus, the built-up list can be analyzed later.

13: Except for the *intial iteration value* of `a`, and the number of loops, this is practically equivalent to items 1, 4, 6, and 9. We can think of this as a little program that takes an initial value of `a` and returns a final value as the last member of the resulting list.

14: We could change the initial value and see how the function varies with its initial value.

15: Or, we can generalize to many initial values, by putting a `Table` and a `For` together. The result is a list of lists (each of length 2): The first entry in each list is the initial value (`dx`) and the second entry is the result of the `For`-loop after four iterations for that `dx`. Because the values tend to get very large, we wrap a `Log` (natural log) around the result of the `For`-loop. A special *increment structure* is utilized—it sets initial and final values as well as the increment size.

3.016 Home

◀◀ ◀ ▶ ▶▶

Full Screen

Close

Quit

Lecture 03 MATHEMATICA® Example 2

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Plotting Lists of Data and Examples of Deeper MATHEMATICA® Functionality

This demonstrates how visualizing data can be combined with other functions to perform analysis. Here, we show that the little iterative program produces a minimum and then we analyze the minimum with two different methods.

```
ListPlot[datatable]
```
1

```
Options[ListPlot]
```
2

*Note that the options are written as **Rules**.*

```
ListPlot[datatable, PlotRange → {250, 500},
 PlotStyle → PointSize[0.025]]
```
3

```
? *Minimum*
```
4

```
FindMinimum[For[a = xvalue;
  i = 1, i ≤ 4, i++, a = 2 a; a = a^a];
 Log[a], {xvalue, 0.15, 0.25}]
```
5

*By going into the Help Browser, you can see that the output of FindMinimum is a list, the first element of which is the functions minimum value, and the second is a **Rule** specifing where the minimum occurs.*

*Lets try and do the above the hard way. I will use **Nest** to recursively apply the function 4 times (I am just using a shorthand here, we can ignore the use of **Nest** for this course...). You can see that it works. Don't worry about it, but if you want to know about it, use the Help Browser to get information about **Nest** and **Pure Functions.***

```
Clear[x]
```
6

```
fx = Nest[(2 #)^(2 #) &, x, 4]
```
7

Take it derivative and set equal to zero...

```
dfx = D[fx, x] // Simplify
```
8

*Finding the zero of this will not be easy.... but **FindRoot** claims it can do it...*

```
FindRoot[dfx, {x, .1, .3}]
```
9

1: The data produced from the last example can be plotted. It is apparent that there is a minimum between initial values of 0.1 and 0.3. But, it will be difficult to see unless the visualization of the plot can be controlled.

3: By specifying the `ListPlot`'s option for the range of the y-like variable, the character of the minimum can be visually assessed.

4: It is likely that MATHEMATICA® has functions to find minima; here we look for likely suspects.

5: `FindMinimum` is a fairly sophisticated function to obtain the minimum of an expression in a specified range, even if the function only returns a numerical result. Here `FindMinimum` is used, to find a very high precision approximation to the minimum observed in item 3. The function is our `For`-loop with a variable `xvalue` as the initial value. We ask `FindMinimum` to hunt for the `xvalue` that minimizes the (`Log` of the) `For`-loop.

7: This is a fairly advanced example—beginning students should not worry about understanding it yet. `Nest` is a sophisticated method of repeated applications of a function (i.e., $f(f(f(x)))$ is nesting the function f three times on an argument x). It is equivalent to the previous methods of producing the iterative structure, but now the result is an expression with a variable `x` that plays the role of the initial value. This concept uses *Pure Functions* which are produced by the ampersand &.

8: The minimum of the function can be analyzed the standard way, here by taking derivatives with `D`. It would not be amusing (that is, for most of us) to find this derivative by hand.

9: `FindRoot` is sophisticated numerical method to obtain the zero of an expression in a specified range.

3.016 Home

◀◀ ◀ ▶ ▶▶

Full Screen

Close

Quit

Very complex expressions and concepts can be built-up by loops, but within MATHEMATICA® the complexity can be buried so that only the interesting parts are apparent and shown to the user.

Sometimes, as complicated expressions are being built up, intermediate variables are used. Consider the value of `i` after running the program:
`FindMinimum[For[a = dx; i = 1, i ≤ 4, i++, a = 2a; a = a∧a]; Log[a], {dx, 0.15, 0.25}]`; the value of `i` (in this case 5) has no useful meaning anymore. If you had defined a symbol such as `x = 2i` previously, then `x` would now have the value of 10, which is probably not what was intended. It is much safer to localize variables—in other words, to limit the scope of their visibility to only those parts of the program that need the variable and this is demonstrated in the next example. Sometimes this is called a "Context" for the variable in a programming language; MATHEMATICA® has contexts as well, but should probably be left as an advanced topic.

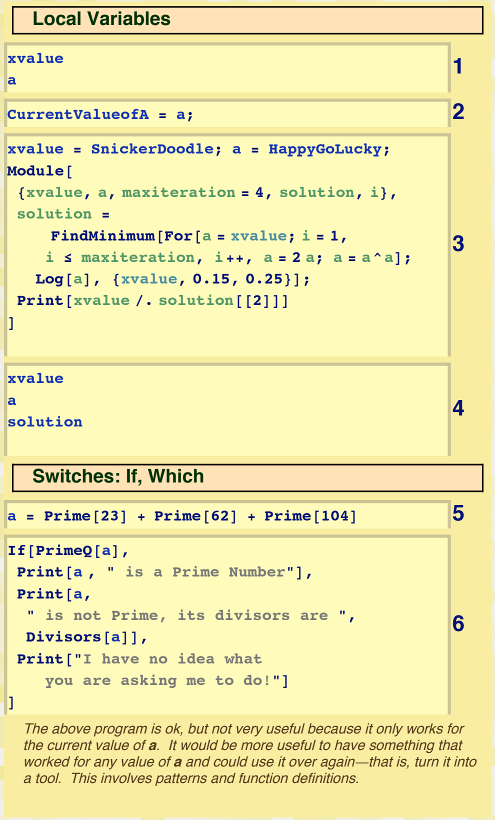
Lecture 03 MATHEMATICA® Example 3

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Making Variables Local and Using Switches to Control Procedures

Describes the use of `Module` to "hide" a variable: consider the variable **a** from the first item in the above example—its intermediate values during iteration are not always important. Suppose you wish to use the symbol **a** later, that it played an intermediate role hence was not used, and may easily be forgotten. It is good practice to make such variables 'local' to their own functions.

An example of a logical switch is demonstrated for `If`.



Local Variables

```
xvalue
a
```
1

```
CurrentValueofA = a;
```
2

```
xvalue = SnickerDoodle; a = HappyGoLucky;
Module[
 {xvalue, a, maxiteration = 4, solution, i},
 solution =
     FindMinimum[For[a = xvalue; i = 1,
    i ≤ maxiteration, i++, a = 2 a; a = a^a];
   Log[a], {xvalue, 0.15, 0.25}];
 Print[xvalue /. solution[[2]]]
]
```
3

```
xvalue
a
solution
```
4

Switches: If, Which

```
a = Prime[23] + Prime[62] + Prime[104]
```
5

```
If[PrimeQ[a],
 Print[a , " is a Prime Number"],
 Print[a,
  " is not Prime, its divisors are ",
  Divisors[a]],
 Print["I have no idea what
    you are asking me to do!"]
]
```
6

*The above program is ok, but not very useful because it only works for the current value of **a**. It would be more useful to have something that worked for any value of **a** and could use it over again—that is, turn it into a tool. This involves patterns and function definitions.*

1: The symbols **xvalue** and **a** are left over from the last example, even though they played only an intermediate role for the final result. It is not unusual to run the same MATHEMATICA® for a day or more—it would be easy to forget that values have been assigned to symbols.

2: This could lead us to mistakenly use its value later as though it might be undefined. This is a common error.

3: The production of such errors can be reduced with a programming practice known as *localized variables* (also known as variable-scoping). The idea is to hide the variable within its own structure—the variable is said to have a limited *scope*. `Module` provides a function for doing this. Here symbols **xvalue** and **a** have set values before the call to `Module`, but any value that is changed *inside* of `Module` has no effect on its "global" value in the rest of the MATHEMATICA® session.. *Using `Module` is good programming practice for creating your own functions.*

4: Even though `Module` changed the symbols **xvalue** and **a**, and used an internal variable **solution**, there should be no effect outside of `Module`.

6: It is useful to build functions that are "smart" (or appear to be so, by applying rules of logic). Here, a simple example of the use of `If` will be applied to a symbol which is the sum of the 23^{rd}, 62^{nd}, and 104^{th} prime numbers.

This is a simple program. First, it checks if **a** is prime using the query-function `PrimeQ`. If the check is true, then it prints a message saying so, and then returns control to the MATHEMATICA® kernel. If the check is false, then it prints out a message and some more useful information about the fact it isn't prime using `Divisors`. If the statement cannot be determined to be true or false, a message to that effect is printed.

3.016 Home

Full Screen

Close

Quit

Patterns are extremely important in mathematics and in MATHEMATICA® . The goal for beginners should be to master how to create your own functions: *understanding how to use patterns is essential to creating your own functions in* MATHEMATICA® .

In MATHEMATICA® , the use of the underscore, _, means "this is a placeholder for something that will be used later."[3] In other words, you may want to perform a predictable action on an object (e.g., find the value of its cosine, determine if it is prime, plot it), but want to create the *action before the object exists*. We create the action using a pattern (_), the arbitrary object, and create fixed operations on the pattern.

Usually, one needs to name the pattern to make it easier to refer to later. The pattern gets named by adding a head to the underscore, such as `SomeVariableName_`, and then you can refer to whatever pattern matched it with the name `SomeVariableName`.

This is a bit abstract and probably difficult to understand without the aid of a few examples. We start with patterns and replacement in the following example, and then build up to functions in the next example.

[3] It is a bit like teaching a dog to fetch—you cock an arm as if to throw _something_, and then when something gets thrown, your dog runs after the "something." The first _something_ is a place holder for an object, say anything from a stick to a ball to the morning paper. The second something is the actual object that is actually tossed, that finally becomes the "something" your dog uses as the actual object in the performance of her ritual response to the action of throwing.

Operating with Patterns

Patterns are identified by the underscore _, and the matched pattern can be named for later use (e.g., `thematch_`).

```
AList = {first, second,
  third = 2 first, fourth = 2 second}
```
1

```
AList /. {2 a_ → a}
```
2

```
Clear[a]
```
3

```
AList /. {2 a_ → a}
```
4

```
AList /.
 {p_ , q_ , r_ , s_} → {p , p q, p q r, p q r s}
```
5

```
{2, 0.667, a / b, Pi} /. {p_Integer → p One}
```
6

*_ all by itself stands for anything. **x_** also stands for anything, but gives anything a name for later use.*

```
AList /. _ → AppleDumplings
```
7

```
PaulieNoMealX = Sum[b[i] x^i, {i, 2, 6}]
```
8

```
PaulieNoMealX /. x^n_ → n x^(n - 1)
```
9

Make the rule work for any polynomial...

```
DerivRule = q_^n_ → n q^(n - 1);
```
10

```
PaulineOMealY = Sum[c[i] z^i, {i, 2, 6}]
```
11

```
PaulineOMealY /. DerivRule
PaulieNoMealX /. DerivRule
```
12

Another problem is that it will not work for first-order and zeroeth-order terms...

```
PaulENoMiel =  Sum[c[i] HoneyBee^i, {i, 0, 6}]
```
13

```
PaulENoMiel /. DerivRule
```
14

This could be fixed, but it would be much easier to do so by defining functions of a pattern.

It is also possible to have a pattern apply conditionally.

```
Cases[{{1, 2}, {2, 1}, {a, b}, {2, 84}, 5},
 {first_, second_} /; first < second]
```
15

1: Construct an example `AList` = {`first, second, 2first, 2second`} to demonstrate use of pattern matching. We will try to replace members that match `2 something` with `something` There is an instructive error in the first try.

2: The rule is applied to `AList` through the use of the operator `/.` (short-hand for `ReplaceAll`). The pattern here is "two multiplied by something." The symbol `a` should a placeholder for *something*, but `a` was already defined and so the behavior is probably not what was wanted: `2 something` was replaced by the current value of `a`. Another (probably better, but better left until later) usage is the *delayed ruleset* `:->`.

4: After `a` has been cleared, the symbol `a` is free to act as a placeholder. In other words, `a` takes on the temporary value of the last match. The effect of applying the rule is 2× *all somethings* are replaced by the pattern represented by `a` which takes a temporary value of *each something*.

5: Here is an example that uses each member of a four-member list, names the members, and then uses a rule to operate on the entire list. Study this example until you understand it.

6: The types of things that get pattern-matched can be restricted by adding a *pattern qualifier* to the end of the underscore. Here, we restrict the pattern matching to those objects that are `Integer`. The first replacement makes sense; however, the third member of the list is understood by considering that the internal representation of a/b is `a`×`Power[b,-1]`—the -1 is what was matched.

7: It is not necessary to name a pattern, but it is a good idea if the match is to be used again later. Here, the first thing that gets matched (the list itself) is replaced with the new symbol.

8: For a simple (incomplete and not generally useful) example of the use of patterns, an example producing symbolic derivative of a polynomial will be developed. Here, a polynomial `PaulNoMealX` in `x` is defined using `Sum`.

9–10: A rule is applied, which replaces patterns `x` to a power with a derivative rule. If only the power is used later, so it is given a place-holder name `n`. This technique would only work on polynomials in `x`. To generalize (**10**), we need a place-holder for the arbitrary variable and its powers.

13–14: This will not work for the constant and linear terms in a polynomial. This could be fixed, but the example becomes complicated and still not as good as MATHEMATICA® 's built-in differentiation rules.

15: To place more control on the types of patterns that get matched, patterns can also be used in conjunction with `Condition` operator `/;`. Here is an example of its use in `Cases`. The pattern is any two-member list *subject to the condition* that the first member is less than the second. Cases returns those members of the list where the pattern was successfully matched.

3.016

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Creating Functions using Patterns and Delayed Assignment

Understanding this example is important for beginners to MATHEMATICA® *!*

The real power of patterns and replacement is obtained when defining functions. Examples of how to define functions are presented.

Defining Functions with Patterns

Defining functions with patterns probably combines the most useful aspects of Mathematica. Define a function that takes patten matching x as its first argument and an argument matching n as its second argument and returns x to the n^{th} power:

```
f[x_ , a_] = x^a;
(*This is not a good way to define
 a function, we will see why later*)
```
1

```
f[2, 3]
f[y, z]
```
2

This works fine, but suppose we had defined ***x*** *ahead of time*

```
x = 4
```
3

```
f[x_ , a_] = x^a;
(*This is not a good way to define
 a function, we will see why later*)
```
4

```
f[2, 3] (*will now be 4^3,
which is probably not what
 the programmer had in mind*)
```
5

```
f[y, z]
```
6

Better Functions with Delayed Assignment (:=)

```
x = 4
a = ScoobyDoo
```
7

```
f[x_ , a_] := x^a
```
8

```
f[2, 5]
```
9

```
f[y, z]
```
10

```
f[x, a]
```
11

```
f[a, x]
```
12

```
Clear[f]
```
13

1: Here is an example of a pattern: a symbol `f` is defined such that if it is called as a function with a pattern of two named arguments `x_` and `a_`, then the result is what ever x^a evaluated to be when the function was defined. **Don't emulate this example—it is <u>not usually</u> the best way to define a function.** In words you are telling MATHEMATICA® , "any time you see f[thing,doodad] replace it with the current value of thing$\hat{\ }$doodad."

2: Our example appears to work, but only because our pattern variables, `x` and `a` had no previous assignment.

3–6 This shows why this can be a bad idea. `f` with two pattern-arguments, is assigned *when it is defined*, and therefore if either `x` or `a` was previously defined, then the definition will permanently reflect that definition. The `=`-assignment is performed immediately and anything on the right-hand-side will be evaluated with their immediate values.

7–12: What we really want to tell MATHEMATICA® in words is, "I am going to call this function in the future. I want to define the function now, but I don't want MATHEMATICA® to evaluate it until it is called; use the pattern-matching variables when you evaluate it later." This involves use *delayed assignment* which appears as `:=`.

For beginning users to MATHEMATICA® , this is the best way to define functions.

In a delayed assignment, the right-hand-side is *not evaluated until the function is called* and then the patterns become *transitory until the function returns its result.* This is usually what we mean when we write $y(x) = ax^2$ mathematically—if y is given a value x, then it operates and returns a value related to that x *and not any other x that might have been used earlier.*

This is the prototype for function definitions.

3.016 Home

Full Screen

Close

Quit

Until you become more familiar with MATHEMATICA®, it is probably a good idea to get in the habit of defining all function with delayed assignment (:=) instead of immediate assignment (=). With delayed assignment, MATHEMATICA® does not evaluate the right-hand-side *until* you ask it to perform the function. With immediate assignment, the right-hand-side is evaluated when the function is defined making it much less flexible because your name for the pattern may get "evaluated away."

Defining functions are essentially a way to eliminate repetitive typing and to "compactify" a concept. This "compactification" is essentially what we do when we define some function or operation (e.g., $\cos(\theta)$ or $\int f(x)dx$) in mathematics—the function or operation is a placeholder for something perhaps too complicated to describe completely, but sufficiently understood that we can use a little picture to identify it.

Of course, it is desirable for the function to do the something reasonable even if asked to do something that might be unreasonable. No one would buy a calculator that would try to return a very big number when division by zero occurs—or would give a real result when the arc-cosine of 1.1 is demanded. Thus, a bit of care is advisable when defining functions: you want them to behave reliably in the future when you have forgotten what you have done. Functions should probably be defined so that they can be reused, either by you or someone else. The conditions for which the function can work should probably be encoded into the function. In MATHEMATICA® this can be done with restricted patterns.

Lecture 03 MATHEMATICA® Example 6

Functional Programming with Recursion: Functions that are Defined by Calling Themselves

This is an example of how one might go about defining a function to return the factorial of a number. Instructive mistakes are introduced and, in the following example, we will make the function behave better with incremental improvements.
We will also show how to speed up programs by trading memory for speed.

The canonical programming example is the factorial function n! = (n)×(n-1) ×(n-2)×...×(1) where 0! ≡ 1; here is a reasonably clever way to use the fact that (n+1)! = (n+1)×n!

```
factorial[n_] := n factorial[n - 1]
```
1

```
factorial[8]
```
2

Ooops, This isn't what was expected, but upon reflection it is correct--we forgot to define a part of the rule. (Note also that the message window produced an error about recursion limits) Add the second part of the definition. Here, we don't use delayed evaluation (:=) because we want to assign a value immediately.

```
factorial[0] = 1;
```
3

```
factorial[120]
```
4

```
factorial[257]
```
5

Here is where the recursion limit comes in : our function keeps on calling itself (i.e., recursively). Unless a limit is set the program would keep running forever. Mathematica builds in a limit to how many times a function will call itself:

```
$RecursionLimit
```
6

```
$RecursionLimit = 2^11
```
7

Speed versus Memory in Functions

```
Timing[factorial[2000]][[1]]
```
8

Using immediate assignment in a function: spending memory to buy time: Each time the function is called, it makes an extra assignment so that previous values can be recalled if needed.

```
factorial[n_] :=
 factorial[n] = n * factorial[n - 1]
```
9

This version takes a bit longer the first time, because we are storing data in memory ...

```
Timing[factorial[2000]][[1]]
```
10

But, the next time it is called, the result is much faster.

```
Timing[factorial[2001]][[1]]
```
11

```
Clear[factorial]
```
12

1: This is a functional definition that will produce the factorial function by recursion because $(n+1)! = (n+1)n!$—the result for $n+1$ is obtained by using the previous result for n.

2: However, trying this function now will produce an advisory in the MATHEMATICA® 's *Message Window*, and will not give a satisfactory result because. . .

3: It is necessary to define a place for the recursion to stop. This is done by *assigning* the factorial of zero to be unity.

5: So that recursive functions don't run for ever, leaving no way, a sensible limit is placed on the number of times a function can call itself. MATHEMATICA® sets a number of variables such as `$RecursionLimit`, that control global behavior.

7: However, the user is free to subvert the defaults.

8: We will now examine the role of memory and speed, to do this we will need the time it takes MATHEMATICA® to do a computation; this can be obtained with `Timing`. `Timing` returns a list of two elements: the first is the time for the computation; the second is the result of the computation. We will only be interested in the first element.

9: Consider using the function to find the factorial of 2000, the currently-defined function must call itself about 2000 times to return a value. Suppose a short time later, the value of 2001! is requested. The function must again call itself about 2000 times, even though all the factorials less than 2001's were calculated previously. If *you* were the CPU, you might say "why are you asking me to do this all again? Can't you remember anything?" Unless computer memory is abundant, it seems like a waste of effort to repeat the same calculations over and over.
Here is an example where computation speed is purchased at the cost of memory. The definition of the function uses a delayed assignment (`:=`) as well as an immediate assignment (`=`). The delayed assignment defines the function with a pattern—the immediate assignment assigns and stores the value of a symbol. Thus, when the function is called, it makes an assignment as well as the computation.

10–11: Here, we see that it takes a little longer to calculate 2000! (because the CPU is doing memory storage operations), but it takes significantly less time to calculate 2001!.

3.016 Home

◀◀ ◀ ▶ ▶▶

Full Screen

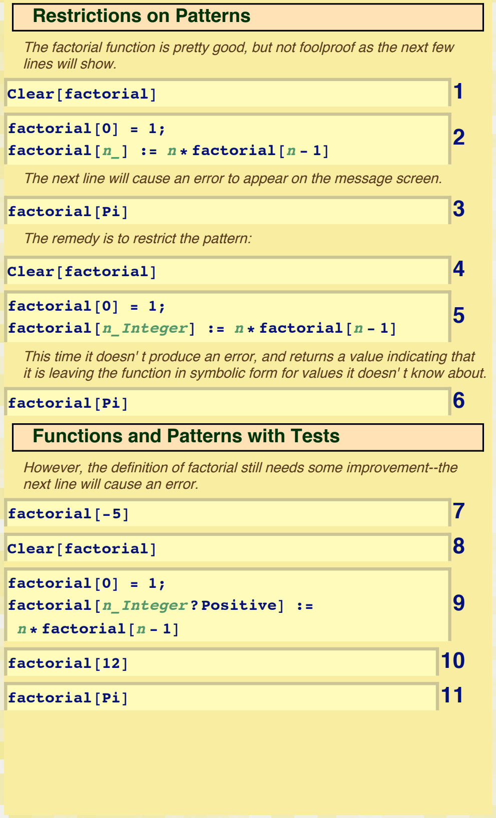
Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Restricted and Conditional Pattern Matching

Here are demonstrations of how to restrict whether a pattern gets matched by the *type of the argument* and how to place further restrictions on pattern matching.



Restrictions on Patterns

The factorial function is pretty good, but not foolproof as the next few lines will show.

```
Clear[factorial]
```
1

```
factorial[0] = 1;
factorial[n_] := n * factorial[n - 1]
```
2

The next line will cause an error to appear on the message screen.

```
factorial[Pi]
```
3

The remedy is to restrict the pattern:

```
Clear[factorial]
```
4

```
factorial[0] = 1;
factorial[n_Integer] := n * factorial[n - 1]
```
5

This time it doesn' t produce an error, and returns a value indicating that it is leaving the function in symbolic form for values it doesn' t know about.

```
factorial[Pi]
```
6

Functions and Patterns with Tests

However, the definition of factorial still needs some improvement--the next line will cause an error.

```
factorial[-5]
```
7

```
Clear[factorial]
```
8

```
factorial[0] = 1;
factorial[n_Integer?Positive] :=
 n * factorial[n - 1]
```
9

```
factorial[12]
```
10

```
factorial[Pi]
```
11

3: However, what if the previously-defined factorial function were called on a value such as π? It would recursively call $(\pi - 1)!$ which would call $(\pi - 2)!$ and so on. Thus, this execution would be limited by the current value of `$RecursionLimit`.
This potential misuse can be eliminated by placing a pattern restriction on the argument of *factorial* so that it is only defined for integer arguments.

5: Here is an improved definition for the *factorial* function using a pattern type: `_Integer`. The type-qualifier at the end of the "_" is the internal representation of whatever the argument was (e.g., `Integer`, `Real`, `Complex`, `List`, `Symbol`, `Rational`, etc.). In this case, the factorial function is *only* defined for integer arguments.

6: Now the function should indicate that it doesn't have anything further to do with a non-integer argument.

7: However, the definition is still not fool-proof because negative integers will not terminate the recursion properly.

9: A pattern can have conditional matching indicated by the `_?Query` where `Query` returns true for the conditions that the pattern can be matched (e.g., `Positive`[2], `NonNegative`[0], `NumberQ`[1.2], `StringQ`["harpo"] all return `True`.) In this example, the function's pattern—`n_Integer?Positive`—might be understood in words as "Match any integer and then test and see if that integer is positive; if so use `n` as a temporary placeholder for that positive integer."

3.016 Home

Full Screen

Close

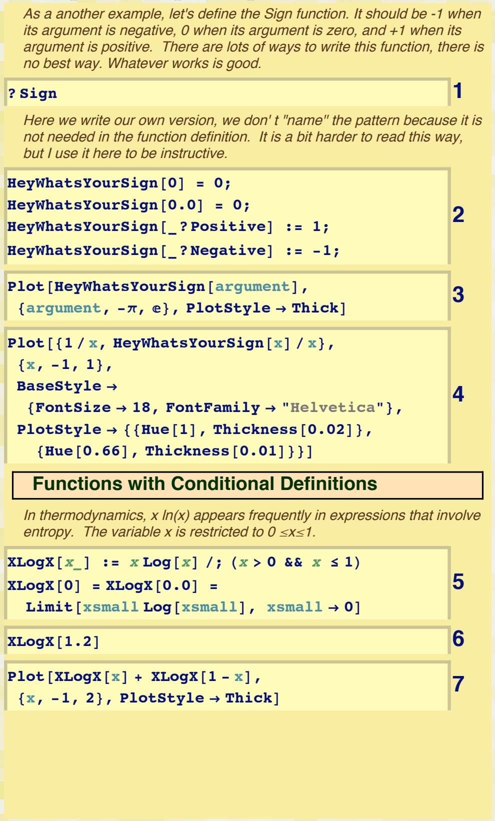
Quit

Lecture 03 MATHEMATICA® Example 8

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Further Examples of Conditional Pattern Matching; Conditional Function Definitions

A simple example of patterns is demonstrated with graphics. Another method of using conditions is demonstrated.



As a another example, let's define the Sign function. It should be -1 when its argument is negative, 0 when its argument is zero, and +1 when its argument is positive. There are lots of ways to write this function, there is no best way. Whatever works is good.

```
? Sign
```
1

Here we write our own version, we don' t "name" the pattern because it is not needed in the function definition. It is a bit harder to read this way, but I use it here to be instructive.

```
HeyWhatsYourSign[0] = 0;
HeyWhatsYourSign[0.0] = 0;
HeyWhatsYourSign[_?Positive] := 1;
HeyWhatsYourSign[_?Negative] := -1;
```
2

```
Plot[HeyWhatsYourSign[argument],
 {argument, -π, ⅇ}, PlotStyle → Thick]
```
3

```
Plot[{1 / x, HeyWhatsYourSign[x] / x},
 {x, -1, 1},
 BaseStyle →
  {FontSize → 18, FontFamily → "Helvetica"},
 PlotStyle → {{Hue[1], Thickness[0.02]},
   {Hue[0.66], Thickness[0.01]}}]
```
4

Functions with Conditional Definitions

In thermodynamics, x ln(x) appears frequently in expressions that involve entropy. The variable x is restricted to 0 ≤x≤1.

```
XLogX[x_] := x Log[x] /; (x > 0 && x ≤ 1)
XLogX[0] = XLogX[0.0] =
  Limit[xsmall Log[xsmall], xsmall → 0]
```
5

```
XLogX[1.2]
```
6

```
Plot[XLogX[x] + XLogX[1 - x],
 {x, -1, 2}, PlotStyle → Thick]
```
7

1: As an example, we will try to duplicate MATHEMATICA® 's definition of `Sign`.

2: Because we want our function to return zero when it gets called with an argument of zero—exact or numerical, immediate assignment is used in the first two lines. (It would probably be better to use the _? `PossibleZeroQ` pattern match here, but slower.)

Because, we don't need to use the value of the matched pattern we can get by without naming it (i.e., `_?Positive`). I include this for instruction purposes—if I were writing this function for later use, I'd probably go ahead and name the pattern for readability.

3: We `Plot` our function to see if it behaves properly. We use `Plot's` option `PlotStyle->Thick` to make the curve easier to see.

4: Here is an example using our function and plotting two curves with more plotting options.

5: The ideal molar entropy of mixing is the sum of $X_i \ln X_i$ for each component i with composition X_i. Because the composition variables are limited to $0 \leq X_i \leq 1$, our example ideal molar entropy function should reflect this constraint.

Here we use a *conditional definition (/;)*, to ensure that our $X \ln X$ function is never called for any X that are out-of-bounds. The delayed assignment statement `LHS := RHS/;test` might be read as, "If the symbol `LHS` is called, then evaluate (using whatever patterns might appear in `LHS`) whether `test` is true; if true, then evaluate RHS with the appropriate pattern replacements." Note that here, we make $X = 0$ a special case and not included in our delayed assignment of the function.

Because $\ln x \to -\infty$ as $x \to 0$, it may not be obvious that $x \ln x \to 0$ as $x \to 0$. We use `Limit` to determine this behavior and use *immediate assignment* in our function definition. (This is a case where immediate assignment makes sense; with delayed assignment the `Limit` function would be called each time that *XLogX* is called on a zero-argument.

7: This is a plot of the ideal molar entropy of mixing for a binary alloy.

3.016 Home

Full Screen

Close

Quit

Index

3.016 Home

Full Screen

Close

Quit