

11.1 Computational Toolbox—Tools of the Trade: *Mathematica* Tutorial

6 (v. 7)

File: *MathematicaTutorial6.nb*

Introduction to Computational Science: Modeling and Simulation for the Sciences
Angela B. Shiflet and George W. Shiflet
Wofford College
© 2006 by Princeton University Press
Tutorial © 2009

Introduction

The prerequisites to this tutorial are *Mathematica* Tutorials 1-5. Tutorial 6 introduces the following functions and options, which simulations of this chapter employ: *Join*, *Length*, *RasterArray*, *Reverse*, *MatchQ*, the blank pattern, two operators (/; and |), *Module*, *Position*, and *x_Integer*. Besides the other commands, Module 11.3 on "Movement of Ants" uses *Position* and *x_Integer*, but Module 11.2 on "Spreading of Fire" does not.

- Do anything that is asked in Quick Review Questions, which appear in cells that look like this one.

Because such cells are text cells and not input cells, do not type in these cells. Instead, move the cursor down until it changes from being vertical to being a horizontal-I bar. Click and start typing; a new input cell will form.

Joining Lists

While the *Mathematica* function *Take* returns a sublist, *Join* returns the concatenation or joining of lists without changing any of the arguments. The following general form returns one list containing the elements from the lists *list1*, *list2*, ...:

```
Join[list1, list2, ...]
```

In the following segment, *Join* concatenates three lists that are a combination of numbers and lists of numbers:

```
Join[{1, 2, 3}, {4, {5}}, {{6, 7}, 8}]
```

- Quick Review Question 1

- a. For the matrix *mat* below, use *Take* to return a list containing the last row. Thus, the returned list is `{{7,8,9}}`.

```
mat = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```

- b. Write a command to return a matrix that is equal to *mat* except the last row of *mat* appears as the first and last rows of the new matrix. Thus, the new matrix has four rows, and its second, third and fourth rows are equal to the rows of *mat*. In your command, do not type specific numbers from *mat* but use function calls to take a sublist and join it to *mat*. Display the answer as a rectangular array of numbers.
- c. Write assignment statements to assign to *listFirstRow* a list containing the first row of matrix *mat* and to assign to *listLastRow* a list containing the last row of *mat*. Then write a command to return the concatenation of *listLastRow*, *mat*, and *listFirstRow*. Thus, for *mat* having three rows, the new matrix has five rows. Display the answer as a rectangular array of numbers.

Length

For generality in functions, it is sometimes useful to obtain a list's length, or the number of elements at the top level of the list. The following form using `Length` returns the number of elements in *expr*:

```
Length[expr]
```

The length returned for each of the lists below is 3. In the second example, the elements are lists.

```
Length[{"a", "b", "c"}]
```

```
Length[{{1, 2}, {0.7, {{{8}}}}, {}]
```

- **Quick Review Question 2** Write a statement to assign to *lst* a list of all zeros that is of random length between 5 and 15 elements. Display the length of *lst*.

Grid Graphics

Frequently, simulations involve evolving rectangular arrays of numbers, and pictorial representations of these matrices can help scientists understand the information. The graphics primitive `Raster` represents such a matrix with each cell being a level of gray or representing an RGB color. With the use of `Graphics`, we can then display the graphics. The form of the `Raster` primitive with a level of gray or RGB triple *gij* for each cell follows:

```
Raster[{{g11,g12,...},{g21,g22,...}...}]
```

The following statement assigns to *matGray* a list of four lists, each consisting of three levels of gray from 0 (black) to 1 (white).

```
matGray = {{0.1, 0.2, 1}, {0, 1, 0.5}, {1, 0, 0}, {1, 1, 0.8}};
```

A call to `TableForm` presents *matGray* as a rectangular array, as follows:

```
TableForm[matGray]
```

With *matGray* as an argument to `Raster`, we display this graphics using `Graphics`, as follows:

```
Graphics[Raster[matGray]]
```

Comparison of the table forms of *matGray* and the picture reveals that the rows appear from bottom to top. Thus, the display for the first row occurs on the bottom, while the last row of the matrix corresponds to the top row of the picture. For simulations, we often do not care about such ordering. However, if visualization of the data is aided by displaying from top to bottom, the order of the rows can be reversed with the `Mathematica` command `Reverse`. In general, the following call reverses the order of the elements in an expression, such as a list:

```
Reverse[expr]
```

The following segment returns the reverse of the list {1, 2, 3}:

```
Reverse[{1, 2, 3}]
```

Reversal occurs at the top level. Thus, the reverse of a list of lists, {*l1*, *l2*, ..., *ln*}, returns the list {*ln*, ..., *l2*, *l1*} without reversing the elements in the list *l1*, *l2*, ..., and *ln*. Thus, the following reverse of {{1,2,3}, {4, 5}} swaps the two elements, {1,2,3} and {4, 5}, without reversing either:

```
Reverse[{{1, 2, 3}, {4, 5}}]
```

Using `Reverse`, we display the graphics corresponding to the rows from top to bottom, as follows:

```
Graphics[Raster[Reverse[matGray]]]
```

In the segment below, we assign a 4-by-3 matrix of zeros and ones to *matEx*. The subsequent assignment for *rgbEx* employs rules to replace each 0 of *matEx* with the graphics triple of RGB (red-green-blue) values for yellow, {1, 1, 0}, and to replace each 1 with the triple for forest green, {0.1, 0.75, 0.2}.

```
matEx = {{0, 0, 1}, {0, 1, 0}, {1, 0, 0}, {1, 1, 0}};
rgbEx = matEx /. {0 -> {1, 1, 0}, 1 -> {0.1, 0.75, 0.2}}
```

Calls to *TableForm* present *matEx* and *rgbEx* as rectangular arrays, as follows:

```
TableForm[matEx]
```

```
TableForm[rgbEx]
```

With *rgbEx* as an argument to *Raster*, we display this graphics using **Graphics**, as follows:

```
Graphics[Raster[rgbEx]]
```

Without the intermediate step of forming *rgbEx*, the following command displays *matEx* with rectangles of yellow and forest green representing 0 and 1, respectively:

```
Graphics[Raster[matEx] /.
  {0 -> {1, 1, 0}, 1 -> {0.1, 0.75, 0.2}}
]
```

Using *Reverse*, we display the graphics corresponding to the rows from top to bottom, as follows:

```
Graphics[Raster[Reverse[rgbEx]]]
```

■ Quick Review Question 3

- a. Generate a 10-by-10 matrix of random floating point numbers between 0 and 1. Display the corresponding graphics with the rows reversed to correspond with the original matrix.
- b. Generate a 10-by-10 matrix of random RGB triples, such as {0.2, 0.1, 0.8}. Display the corresponding graphics.
- c. Generate a 10-by-10 matrix *mc* of RGB triples with the first and second coordinates each varying from 0 to 0.9 in steps of 0.1 and with the third coordinate being 0. Display the corresponding graphics with an *Automatic* aspect ratio.
- d. Smooth out the display with a 100-by-100 matrix *mc2* and appropriately smaller step sizes.

Matching Patterns for Definitions

Frequently, a *Mathematica* function that drives a simulation has different definitions for various configurations of the arguments. We can use the operator `/;` to activate a definition if a test is true and *MatchQ* to perform the test that the arguments match a particular configuration.

The following general call to *MatchQ* returns *True* if the pattern *form* matches the expression *expr*; and otherwise, *MatchQ* returns *False*:

```
MatchQ[expr, form]
```

In the following segment, *MatchQ* returns *True* because $15z$ with z equal to 2 matches the pattern 30:

```
z = 2;
MatchQ[15 z, 30]
```

We can use the "or" operator `|` to generate a pattern that represents any of several choices. Thus, for patterns $p1$, $p2$, and $p3$, the following expression matches $p1$ or $p2$ or $p3$:

```
p1 | p2 | p3
```

The following command verifies that $15z$ matches the pattern 30 or 40:

```
MatchQ[15 z, 30 | 40]
```

To write a *Mathematica* program with several branches, we can employ nested calls to *If*. However, using several definitions for a function, we can make the program easier to correct, modify, and understand. (The ability to have several definitions for the same function name is the feature **polymorphism** in object oriented languages, such as C++ and Java.)

When a function has alternate definitions, *Mathematica* employs the definition with the most specific pattern of the parameters that matches the arguments. For example, the most general definition for the function *exFnc* below returns 3 times the

argument. However, when the argument is 5, the function returns 2. The graph of this function is a straight line with a hole at $x = 5$ and the point $(5, 2)$. As the calls to the function indicate, when we use any argument other than 5, the first definition applies; but with an argument of 5, *Mathematica* employs the second definition with its more specific match of the pattern 5.

```
Clear[exFnc]
exFnc[x_] := 3 x
exFnc[5]_ := 2
```

```
exFnc[4]
```

```
exFnc[5]
```

In one kind of random walk, depending on the value of a random integer, the next point in a path might be the current site or one step in any N, E, S, or W direction. With r storing a random number, the nested *If* calls below return the next point in a random walk. This point is an ordered pair with no change or an increment or decrement by 1 of x or y , based on the value of r .

```
If[r == 0, {x + 1, y}, (* east *)
  If[r == 1, {x - 1, y}, (* west *)
    If[r == 2, {x, y + 1}, (* north *)
      If[r == 3, {x, y - 1}, (* south *)
        {x, y}]]]]
(* site *)
```

The following function definitions for *dirFnc* generate the same logic. Subsequent calls to the function illustrate its action. Depending on the value of the first argument, the appropriate definition is invoked.

```
Clear[dirFnc]
dirFnc[0, x_, y_] := {x + 1, y} (* east *)
dirFnc[1, x_, y_] := {x - 1, y} (* west *)
dirFnc[2, x_, y_] := {x, y + 1} (* north *)
dirFnc[3, x_, y_] := {x, y - 1} (* south *)
dirFnc[r_, x_, y_] := {x, y}
(* site *)
```

With the first argument being 2 in the following call to *dirFnc*, the "north" definition applies to return the point to the north of $\{17, 35\}$:

```
dirFnc[2, 17, 35]
```

The following call to *dirFnc* with a first argument of 5 invokes the most general definition to return the site designated by the second and third arguments:

```
dirFnc[5, 17, 35]
```

The last definition above is as follows:

```
dirFnc[r_, x_, y_] := {x, y} (* site *)
```

The definition does not use the first parameter r . Thus, we do not need to name that parameter and can use an underscore (`_`), which indicates a **blank pattern** that matches any *Mathematica* expression, as follows:

```
dirFnc[_ , x_, y_] := {x, y} (* site *)
```

The **condition operator** `/;` enables us to have alternative function definitions depending on the pattern of the arguments. The following definitions of f return the absolute value of nonzero arguments. Thus, f returns any positive argument. However, if an argument is less than zero, f returns the negative of the argument. As the last call to f reveals, the function is undefined at 0.

```
Clear[f]  
f[x_] := x /; x > 0  
f[x_] := -x /; x < 0
```

```
f[3]
```

```
f[-3]
```

```
f[0]
```

- Quick Review Question 4** Define a function *g* using */;* and *MatchQ* with alterative definitions, not *If*. The function has three parameters, *n*, *a*, and *b*. With a first argument of 1, if *a* or *b* is 3, the function returns *a* + 1. With a first argument of 2, if *a* or *b* is 4, the function returns *b* + 1. Otherwise, with a blank first parameter, the function returns *a* + *b*. Test the function thoroughly.

Local Variables

In longer function definitions, we often employ temporary variables that should only be active within the definition. To avoid conflicts with variable names elsewhere, we make the variables local to the definition with `Module`. A general form of a call to `Module` is as follows:

```
Module[{x, y, ...}, expr]
```

The list of local variables, `{x, y, ...}`, is followed by a comma and an expression, *expr*. Semicolons separate commands in the expression.

In the definition of *diag* below to display a diagonal of green squares in an *n*-by-*n* yellow field, we employ local variables *matn* and *rgbmatn*. The local variable *matn* stores an *n*-by-*n* matrix of zeros with ones on a diagonal. After a semicolon, the assignment to *rgbmatn* replaces 0 with the RGB triple to designate yellow and 1 with the RGB values for forest green. The command to generate and show the raster array occurs after another semicolon.

```
Clear[matn, rgbmatn, diag]
diag[n_] := Module[{matn, rgbmatn},
  matn = Table[If[i == j, 1, 0], {i, n}, {j, n}];
  rgbmatn = matn /. {0 -> {1, 1, 0}, 1 -> {0.1, 0.75, 0.2}};
  Graphics[Raster[Reverse[rgbmatn]], AspectRatio -> Automatic]
];
```

A call to *diag* with argument 4 displays a 4-by-4 yellow square with green squares on a diagonal.

```
diag[4]
```

However, outside the function definition, variables *matn* and *rgbmatn*, which are local to *diag*, do not retain their local values.

```
matn
rgbmatn
```

- Quick Review Question 5** In Quick Review Question 3 a, you generated a 10-by-10 matrix of random floating point numbers between 0 and 1 for the coordinates and displayed the corresponding grayscale graphics. Define a function *randPic* with no parameters to perform these tasks. Have a

local variable *mat* to store the 10-by-10 matrix. Invoke the function several times to observe the changing graphics.

Mathematica does not allow us to change the value of a parameter within a function. For example, the following function incorrectly attempts to change the value of the parameter *x* and return double the new value.

```
Clear[incorrect];
incorrect[x_] := Module[{},
  x = x + 1;
  2 x
]
```

Execution of the segment below does not change the value of the argument (*z*) and so returns double the argument's original value, 3. Because a function cannot change the value of an argument, *Mathematica* generates an error message.

```
z = 3;
incorrect[z]
z
```

Alternatively, we define a correct function with local variable, *localVar*, that becomes one more than the parameter, *x*. The function returns double *localVar*.

```
Clear[correct];
correct[x_] := Module[{localVar},
  localVar = x + 1;
  2 localVar
]
```

A call to *correct* with argument 3 does correctly return $2(3 + 1) = 8$.

```
z = 3;
correct[z]
```

Position of a Pattern

The *Mathematica* function `Position` identifies the locations of pattern, *pattern*, in a list, *list*, with the following command form:

```
Position[list, pattern]
```

The function returns a list with each index in a separate list. We can use these indices in further computations, such as finding corresponding values in two lists.

The following call to *Position* returns a list of lists indicating the indices of *b* (2, 4, and 5) in $\{a, b, c, b, b\}$:

```
pos = Position[{a, b, c, b, b}, b]
```

To eliminate the braces around the individual indices, yielding $\{2, 4, 5\}$, we can employ the following call to the function *Flatten*:

```
Flatten[pos]
```

■ Quick Review Question 6

- a. Assign to *ages* a list of 10 random integers with values between 21 and 23, inclusively. For example, *ages* might be a list of the ages of 10 patients. Then, using *Position* and *Flatten*, give a command to assign to *pos21* a list of the indices where 21 occurs. This list should contain integers, not lists of integers.
- b. Assign to *weight* a list of 10 random integers between 100 and 250. For example, *weight* might store the weights of 10 patients. Using *pos21* from Part a, *weight*, and double brackets (`[[]]`), write an expression to display the weights of the patients who are 21 years old.

Integer Parameters

To indicate that a parameter, such as *x*, is of a certain type, such as *Integer*, we enter the type after the parameter name and an underscore, such as `x_Integer`. For example, consider the following list of ordered pairs:

```
lst = {{3, 14}, {6, 15}, {-2, 7}};
```

To obtain a list of all first coordinates, we apply the following rule to transform each ordered pair with an integer first coordinate into that coordinate:

```
lst /. {x_Integer, _} -> x
```

Without the *Integer* designation, *Mathematica* matches the pattern at a higher level to return the first ordered pair, as output from the following command indicates:

```
lst /. {x_, _} -> x
```

- **Quick Review Question 7** Define a function, h , with one parameter, x , that only accepts integer arguments and returns 2^x . Illustrate that the function returns the proper value for integer arguments 3, -3, and 0 but has no definition for arguments of other types, such as 0.3 and 1/2.