0 Introduction

Preface

This book is intended to supplement Ken Rosen's *Discrete Mathematics and Its Applications, Eighth Edition*, published by McGraw-Hill. It was developed with *Mathematica 11*, created by Wolfram Research. This is intended to be a guide as you explore concepts in discrete mathematics and to provide you with tools you can use to investigate further on your own. This text can significantly enhance a traditional course in discrete mathematics in several ways. First, it makes a plethora of examples readily available that you can interact with easily. Second, it makes the notion of algorithm, which is central in discrete mathematics, concrete by giving you the opportunity to actually implement algorithms rather than only analyze them in the abstract. Finally, and most significantly, it provides you greater freedom to make conjectures and experiment without getting bogged down in repetitive calculation.

The focus of this manual is on Wolfram Language code and does not attempt to explain discrete mathematics. It is expected that you are taking, or have taken, a course in discrete mathematics. Ideally, you have access to Ken Rosen's *Discrete Mathematics and Its Applications*. It is not assumed that you have any prior experience with *Mathematica* or the Wolfram Language. The introductory chapter that follows this preface is designed to introduce you to *Mathematica* and the Wolfram Language. Likewise, it is not assumed that you have any experience with computer programming languages. Part of the Introduction is devoted to the basic concepts and techniques of computer programming. Subsequent chapters gradually introduce increasingly sophisticated programming ideas. While this is not a textbook on computer programming, you will likely find yourself fairly comfortable with the basics of programming by the end.

With the exception of the Introduction, the structure of this book follows that of *Discrete Mathematics and Its Applications*. For each section of each chapter in that text, this manual contains a corresponding section describing built-in Wolfram Language functions and developing additional functions that are used to explore the mathematics topics in that section. Each chapter also contains solutions to some of the Computer Projects and Computations and Explorations exercises found at the end of the chapter of *Discrete Mathematics and Its Applications*. You will also find a number of exercises at the conclusion of each chapter designed to suggest additional questions that you can explore.

This manual strikes a balance between describing existing Wolfram Language functions and creating new procedures that extend *Mathematica*'s capabilities for exploring discrete mathematics. For example, the Wolfram Language does not explicitly support calculations with ordered graphs. Therefore, in Chapter 10, in addition to describing the Wolfram Language's capabilities for modeling graphs, we also write functions relating to ordered graphs. Some readers may not be interested in the detailed descriptions of how new functions and programs like these are created. However, even if you are not interested in the programming aspect, the functions we create are still available to you to explore those topics.

This manual is based on *Exploring Discrete Mathematics with Maple* and retains the spirit and goals of that work. We therefore reproduce the preface of the original book below.

Changes in the New Edition

The previous version of this manual was written for the seventh edition of Ken Rosen's *Discrete Mathematics and Its Applications* and developed with *Mathematica* version 9. The current version includes significant revisions and updates to reflect the revisions in the eighth edition of the textbook and the improvements present in version 11 of *Mathematica*. Some of the most notable revisions include:

- Exposition and programming examples have been improved, with a focus on simplifying and clarifying both to help you more easily understanding connections to the mathematics content.
- Functions defined in each chapter are now provided online as plain text source files with the extension .wl. These files can be imported into a *Mathematica* or other Wolfram System session, or they can be opened with any other software capable of viewing plain text files.
- Additional examples have been added to reflect new content in the eighth edition of *Discrete Mathematics and Its Applications*, including solving the *n*-Queens problem via satisfiability, implementing the naive string matching algorithm, illustrating homomorphic encryption, and exploring semantic networks.
- Wolfram Language functions that have been improved or added to the Language since the last version of the manual have been incorporated. In some cases, most prominently with regard to graph theory, the improvements to the built-in functions have made it no longer necessary to develop functions within this manual to fill gaps in the Wolfram Language. Several "from scratch" functions duplicating built-in capabilities remain when doing so illustrates important mathematics content or programming techniques.
- In version 10, Associations were added to the Wolfram Language. This is an important and fundamental data structure and has been incorporated throughout the manual, largely replacing the need for indexed objects and downvalues. Associations are introduced in Chapter 2 to represent fuzzy sets and functions on finite domains. The related structure Dataset was also added in version 10 and is described in relation to relational databases.
- Users of the previous edition of this manual should be aware that some of the improvements in the Wolfram Language have resulted in sometimes subtle changes to the functions defined in the text. For example, SubsetQ was added to the Wolfram Language in version 10. The predicate defined in the previous version of the manual to fill that gap in the language has been retained as an example of programming control structures (a Do loop with Catch and Throw). However, the manual's function was revised to match the order of arguments in the built-in function.

Acknowledgments

I am deeply grateful to Ken Rosen for having trusted me with this work and for his wisdom and guidance. I am indebted to the authors of the original *Exploring Discrete Mathematics with Maple* for providing an excellent foundation on which to build.

I also wish to thank Nora Devlin, the Product Developer at McGraw-Hill Higher Education for *Discrete Mathematics and Its Applications*, eighth edition, for her patience and confidence.

Thanks also to those who have provided feedback on the previous version.

I am grateful to Martin Erickson for his mentorship. To Daniel Baack, Jason Beckfield, Elizabeth Davis-Berg, Julie Minbiole, Christopher Shaw, Michael Welsh, and Heather Minges Wols for their constant support and encouragement. Finally, I am always grateful to my parents for all they have done.

Daniel R. Jordan
djordan@colum.edu

Preface to the First Edition of *Exploring Discrete Mathematics with Maple*

This book is a supplement to Ken Rosen's text *Discrete Mathematics and Its Applications, Third Edition*, published by McGraw-Hill. It is unique as an ancillary to a discrete mathematics text in that its entire focus is on the computational aspects of the subject. This focus has allowed us to cover extensively and comprehensively how computations in the different areas of discrete mathematics can be performed, as well as how results of these computations can be used in explorations. This book provides a new perspective and a set of tools for exploring concepts in discrete mathematics, complementing the traditional aspects of an introductory course. We hope the users of this book will enjoy working with it as much as the authors have enjoyed putting this book together.

This book was written by a team of people, including Stan Devitt, one of the principle authors of the Maple system, and Eithne Murray, who has developed code for certain Maple packages. Two other authors, Troy Vasiga and James McCarron, have mastered discrete mathematics and Maple through their studies at the University of Waterloo, a key center of discrete mathematics research and the birthplace of Waterloo Maple Inc.

To effectively use this book, a student should be taking, or have taken, a course in discrete mathematics. For maximum effectiveness, the text used should be Ken Rosen's *Discrete Mathematics and Its Applications*, although this volume will be useful even if this is not the case. We assume that the student has access to Maple, Release 3 or later. We have included material based on Maple shareware and on Release 4 with explicit indication of where this is done. (Where to obtain Maple shareware is described in the Introduction.) We do not assume that the student has previously used Maple. In fact, working through the book can teach students Maple while they are learning discrete mathematics. Of course, the level of sophistication of students with respect to programming will determine their ability to write their own Maple routines. We make peripheral use of calculus in this book. Although all places where calculus is used can be omitted, students who have studied calculus will find this material of interest.

This volume contains a great deal of Maple code, much based on existing Maple functions. But substantial extensions to Maple can be found throughout the book; new Maple routines have been added in key places, extending the capabilities of what is currently part of Maple. An excellent example is new Maple code for displaying trees, providing functionality not currently part of the network package of Maple. All the Maple code in this book is available over the Internet; see the Introduction for details.

This volume contains an Introduction and 13 Chapters. The Introduction describes the philosophy and contents of the chapters and provides an introduction to the use of Maple, both for computation and for programming. This chapter is especially important to students who have not used Maple before. (More material on programming with Maple is found throughout the text, especially in Chapters 1 and 2.) Chapters 1 to 10 correspond to the respective chapters of *Discrete Mathematics and Its Applications*. Each chapter contains a discussion of how to use Maple to carry out computation on the subject of that chapter. Each chapter also contains a discussion of the Computations and Explorations found at the end of the corresponding chapter of *Discrete Mathematics and Its Applications* along with a set of exercises and projects designed for further work.

Users of this book are encourage to provide feedback, either via the postal service or the Internet. We expect that students and faculty members using this book will develop material that they want to share with others. Consult the Introduction for details about how to download Maple software associated with this book and for information about how to upload your own Maple code and worksheets.

Introduction

Modern mathematical computation systems, such as *Mathematica* and other products implementing the Wolfram System, allow us to carry out complicated computations quickly and easily. As a supplement to traditional exercises solved by hand, having computational tools available while learning discrete mathematics provides a new dimension to the learning experience. Specifically, computational tools support an inquiry and experimental approach to learning. This book is designed to connect the traditional approach to learning discrete mathematics with this experimental approach.

Using computational software, students can experiment directly with many objects that are important in discrete mathematics. These include sets, large integers, combinatorial objects, graphs, and trees. Furthermore, by using interactive computational software, students can explore these examples more thoroughly, fostering a deeper understanding of concepts, applications, and problem-solving techniques.

This manual has two main goals. The first is to help students learn how to model and solve problems in discrete mathematics using the Wolfram Language. The second is to be a guide and a model as students discover mathematics with the use of computational tools.

This book is intended for use by any student of discrete mathematics. No previous familiarity with the Wolfram Language or *Mathematica* is required. Likewise, we do not assume any previous experience with computer programming. The fundamentals of the Wolfram Language and the basic concepts of computer programming will be thoroughly explained as they are needed.

This manual was created within the desktop version of *Mathematica*, which is one of the products that implement the Wolfram System. Roughly, the Wolfram System is the computational engine and the Wolfram Language is the programming language and built-in functions that are evaluated by the System. This manual will teach you the basics of the Wolfram Language, and there are numerous products created by Wolfram Research that you can use, including *Mathematica* (both desktop and online), Wolfram Programming Lab, and Wolfram Development Platform. For the sake of brevity, this manual will use the word *Mathematica* to refer to whatever particular product you may be using.

Structure of This Manual

This supplement begins with a brief introduction to *Mathematica*, its capabilities, and its use. The material in this introductory chapter explains the philosophy behind working with *Mathematica*, how to use *Mathematica* to carry out computations, and its basic structure. This introduction continues by explaining the basic concepts and syntax for programming with the Wolfram Language. This will provide those who are new to the Wolfram Language in particular and programming languages in general the background that they will need in the rest of the book.

Following the introduction, the main body of this book contains 13 Chapters. Each chapter parallels a chapter of *Discrete Mathematics and Its Applications, Eighth Edition*, by Kenneth H. Rosen (henceforth referred to as the text or the textbook). Each chapter includes comprehensive coverage explaining how the Wolfram Language and *Mathematica* can be used to explore the topics of the corresponding chapter of the text. This includes a discussion of relevant Wolfram Language functions, many new functions that are written expressly for this book, and examples illustrating how to use *Mathematica* to explore topics in the text.

Additionally, we discuss selected *Computer Projects* and *Computations and Explorations* from the corresponding chapter of the text. We provide guidance, partial solutions, or complete solutions to these exercises. A similar philosophy governs the inclusion of these solutions as does the inclusion of answers

to selected exercises in the back of most mathematics textbooks. You should attempt the problem on your own first. The solutions in this manual are intended to be referred to: after you have succeeded in solving a problem to see a (potentially) different approach; when you have stopped making progress on your own and need a slight boost to continue; or when you are trying to solve a similar problem.

Finally, each chapter concludes with a set of additional questions for you to explore. Some of these are straightforward computational exercises, whereas others are more accurately described as projects requiring substantial additional work, including programming.

The chapters of this manual are available in two formats: as a PDF document and as a *Mathematica* Notebook. The PDF format contains all of the text and *Mathematica* functions and other information that you need. The *Mathematica* Notebook version of the chapter includes additional features, specifically active *Mathematica* code and links to *Mathematica* documentation. If you are using the desktop version of *Mathematica*, it is recommended that you primarily work with the *Mathematica* Notebook version of this manual. However, the Notebooks are quite large and some of the elements may not work well with the cloud-based systems, in which case the PDF version may be the better option. Even so, the definitions of the functions created within the text of the manual are available as simple .wl files, which can be uploaded to the cloud, so that you do not need to retype the code. Instructions for how to do this is provided near the end of this chapter.

When you first open the *Mathematica* Notebook for any chapter, it is recommended that you evaluate the initialization cells in that notebook by selecting **Evaluate Initialization Cells** from the **Evaluation** menu. This way, the essential symbols within the chapter, those associated with variables and functions you may want to use, will be available for you, without your having to execute each of their definitions manually.

If you do not explicitly cause the initialization cells to be evaluated, the first time you evaluate any cell in the notebook, you will see a dialog box asking whether you wish to evaluate the initialization cells. Selecting "Yes" is recommended and will cause all of the initialization cells to be executed.

Alternatively, selecting **Evaluate Notebook** from the **Evaluation** menu will cause every input cell in the notebook to be evaluated. If you choose this option, you will also see a message asking whether or not you want to evaluate all of the initialization cells. In this case, having selected **Evaluate Notebook**, it is recommended you choose "No" in the dialog. Selecting "Yes" will cause all of the initialization cells to be evaluated and then every cell in the notebook will be evaluated, meaning the initialization cells would each be evaluated twice, which is unnecessary.

For some chapters, you may also see a warning that the file "contains potentially unsafe dynamic content." This will appear when the file contains an animation or an interactive element. If you see this warning, it is recommended that you click on the button to "Enable Dynamics" in order to fully engage with the content.

The main benefit of the *Mathematica* Notebook version of this book is that it is interactive. That is, you can execute the Wolfram Language functions demonstrated in the chapter. Even better, you can modify the examples so that you can experiment right within the body of the chapter. Additionally, you have immediate access to the Wolfram Language help and documentation pages. Within the text of this manual, Wolfram Language functions appear in blue and are underlined indicating that clicking on them will open the corresponding documentation page.

This book has been designed to help students achieve the main goals of a course in discrete mathematics. These goals, as described in the preface of the textbook, are the mastery of mathematical thinking, combinatorial analysis, discrete structures, algorithmic thinking, and applications and modeling.

This supplement demonstrates how to use the interactive computational environment of *Mathematica* to enhance and accelerate the achievement of these goals.

Interactive *Mathematica*

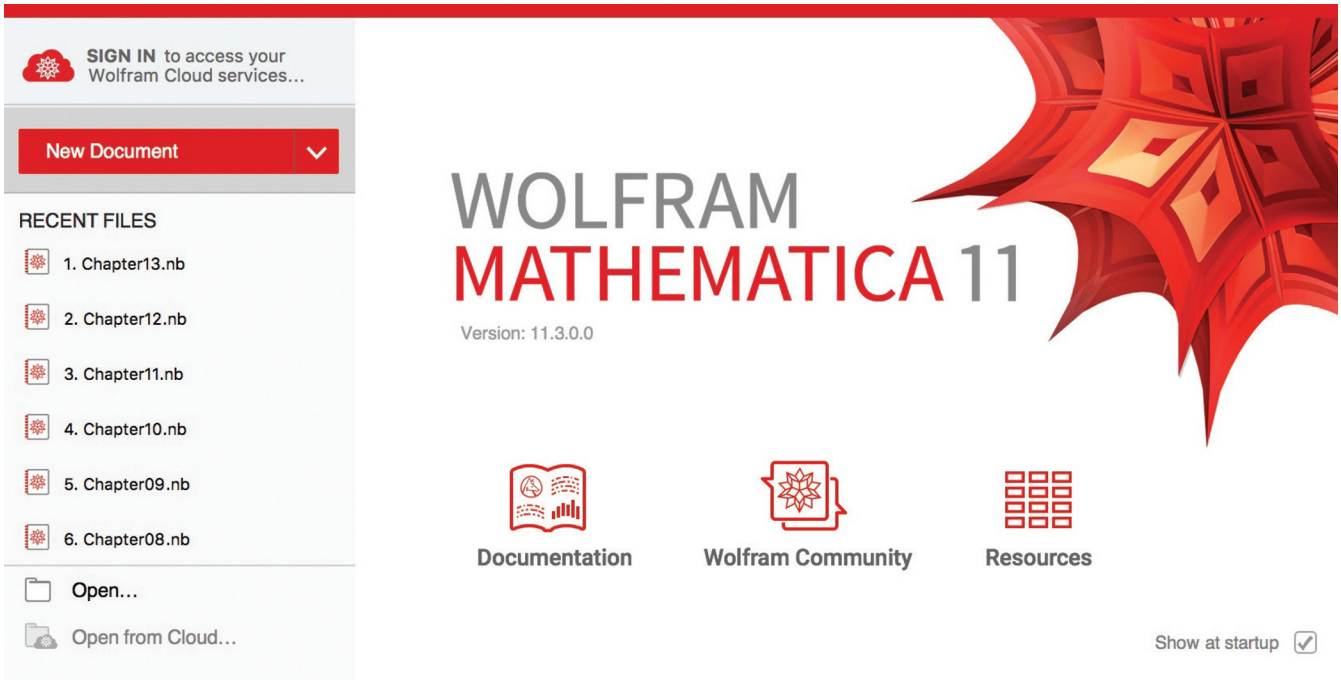
Exploring discrete mathematics with *Mathematica* is like exploring a mathematical topic with an expert assistant at your side. As you investigate a topic, you should always be asking questions. In many cases, the answer to your question can be found by experimenting. *Mathematica*, your highly trained mathematical assistant, can often carry out these directed experiments quickly and accurately, often with only a few simple instructions.

By hand, the magnitude and quantity of work required to investigate even one reasonable test case may be prohibitive. By delegating the details to *Mathematica*, your efforts can be much more focused on choosing the right mathematical questions and on interpreting results. Moreover, with a system such as *Mathematica*, the types of objects you are investigating, and tools for manipulating them, already exist as part of the basic infrastructure provided by the system. This includes lists, variables, polynomials, graphs, arbitrarily large integers, rational numbers, and most important, support for exact and fast computations.

The use of *Mathematica* is merely a means to the end of achieving the goals of a course in discrete mathematics. As with any tool, to use it effectively you must have some basic understanding of the tool and its capabilities. In this section, we introduce *Mathematica* by working through a sample interactive session.

Starting *Mathematica*

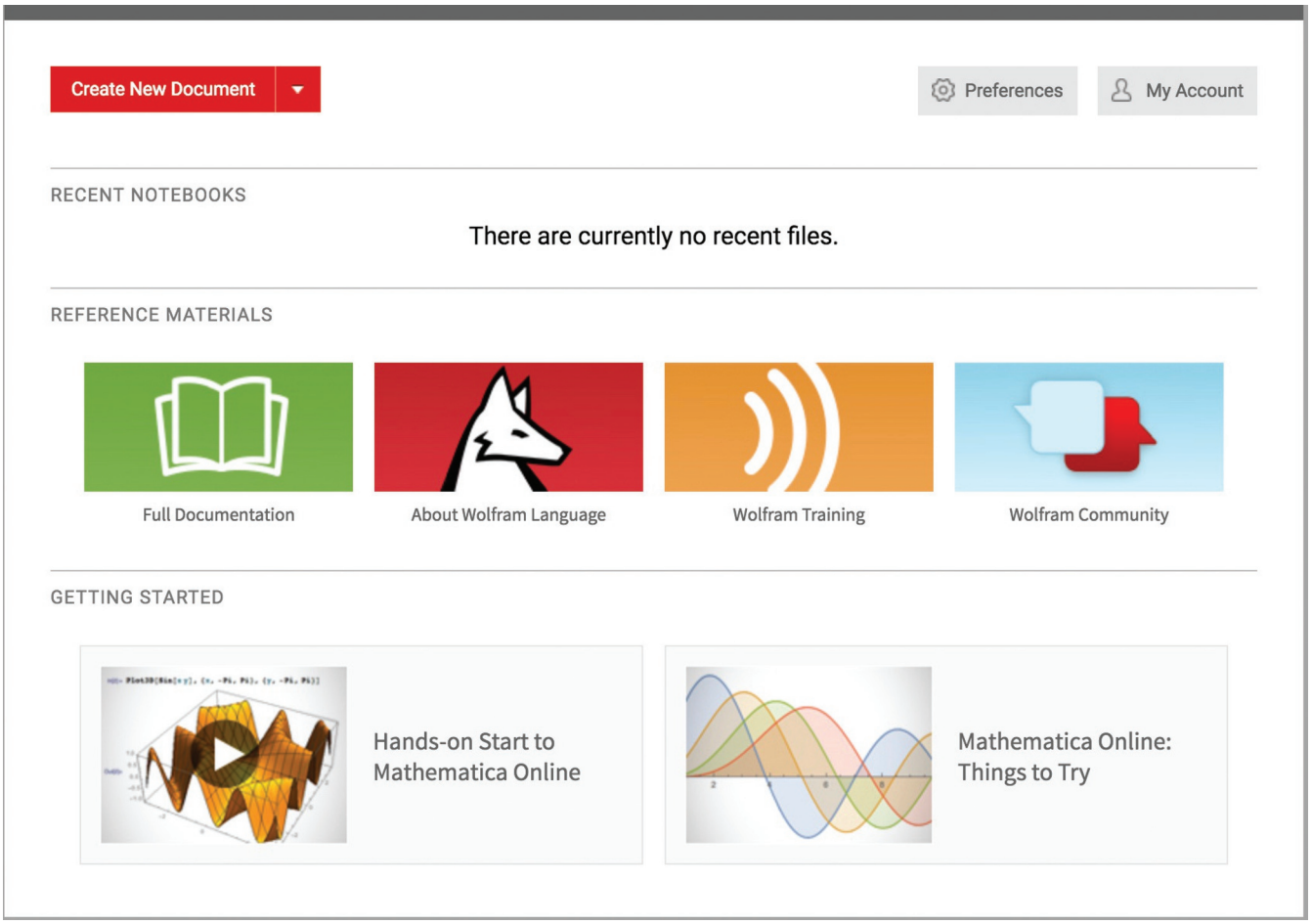
A new *Mathematica* session begins when you start the *Mathematica* software or log in to the web interface. If you are using the desktop version of *Mathematica*, you may see a welcome screen like the one below.



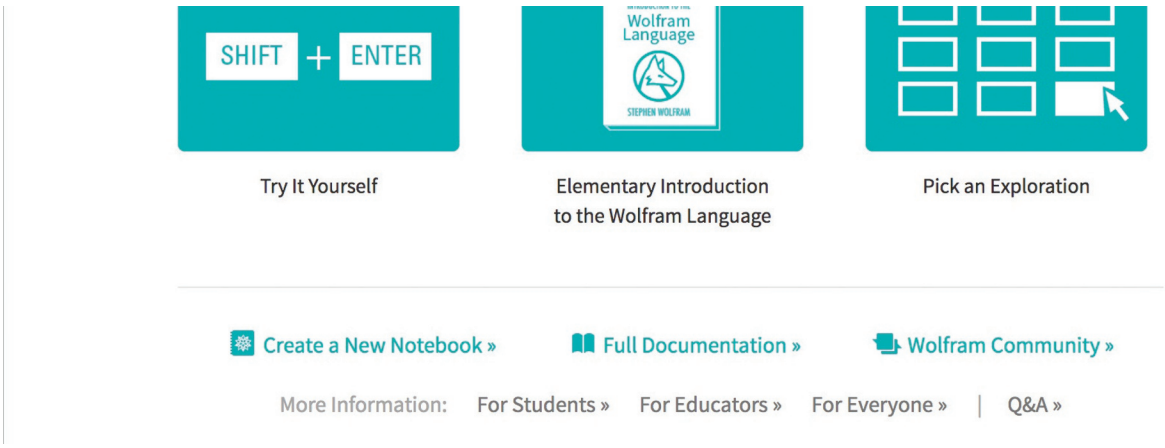
The central portion of the welcome screen welcomes you to *Mathematica* and provides links to documentation, support, and demonstrations.

On the left side of the welcome screen are icons for creating new documents or opening existing documents. To open an existing file that is not in the list of recent documents, perhaps a notebook that you created or one of the chapters of this manual, you click on "Open..." and a standard file selection dialog will appear that allows to you select the file you want. Otherwise, to create a new document to work in, simply click on "Notebook" under "New Document" at the top of the left panel. If the welcome screen does not appear, you can open existing and create new Notebooks using the File menu items.

If you are using *Mathematica* Online or Wolfram Programming Lab or another one of the online systems, when you log in, you may also be presented with a welcome screen, such as the one below, with an option to create a new document. Again, clicking on the red button will reveal additional options, and you should select Notebook. You may also have the option to upload existing Notebook files.



Using the Wolfram Programming Lab in the Open Cloud, you may need to scroll to the bottom of the welcome window to find the Create a New Notebook link.



Mathematica Notebooks

All the chapters of this manual were created as *Mathematica* Notebooks. A Notebook can be thought of like a document in a word processing program. You can type text, change the font, insert images, and use other typical word processing tasks. However, you also have a powerful mathematical engine at your fingertips.

In a Notebook, everything you enter and all of the results of computations are stored in cells. We will mention three kinds of cells: input, output, and text.

To create a new cell, use the mouse or keyboard arrow keys to the bottom of a Notebook or between two existing cells. You should see a faint horizontal line across the entire window with a plus sign on or below the line at the left margin.

By clicking on the plus symbol, *Mathematica* will present you with a menu of the most common cell types. To create an input cell, click on "Wolfram Language input", or click on "Plain Text" to create a text cell.

Alternately, you can press the command key (the cloverleaf on a Mac) and the number 7 (CMD+7 or ⌘+7) to create a text cell. For an input cell, press CMD+9 or ⌘+9.

Finally, if you simply start typing, *Mathematica* will automatically create a new cell for you. The kind of cell depends on options set in the software and on the particular Notebook's style, but for a brand new Notebook with the default options, an input cell should be created if you start typing with the cursor not in an existing cell.

Evaluating Expressions

To evaluate a mathematical expression, first make sure that the cursor is somewhere in the input cell containing the expression you wish to evaluate. Then, press shift together with the enter or return key on the keyboard. Alternatively, if your keyboard includes a numeric keypad, you can press the enter key alone on the keypad. Try this with the input below.

```
In[1]:= 2+3
Out[1]= 5
```

Here are a few more expressions. Try entering them on your computer.

```
In[2]:= Sum[i^2,{i,1,10}]
Out[2]= 385
In[3]:= Integrate[(x-1)^3,x]
Out[3]= 1/4 (-1+x)^4
In[4]:= Expand[%]
Out[4]= 1/4 - x + 3 x^2/2 - x^3 + x^4/4
```

The percent symbol, referred to in the Wolfram Language as `Out (%)`, is used to refer to the contents of the most recent output cell. It does not always refer to the value immediately above it, as evaluation can be done out of order.

The line numbers, which are automatically attached to the input and output cells, can be used to refer to specific results by following the `Out (%)` with the line number. For example, to use the value from output line 3, you would enter `%3`.

```
In[5]:= %3+1
Out[5]= 1 + 1/4 (-1+x)^4
```

Note that `Out (%)` is useful when working interactively, but line numbers change every time you evaluate cells. In particular, if you save a Notebook and quit *Mathematica* and then reopen the same Notebook later, the line numbers may not refer to the same expressions, particularly if you do not evaluate the same cells in the same order. We tend to avoid using `Out (%)` in this manual, but you should keep it in mind for your own computations.

A First Encounter with *Mathematica*

As already indicated, working with *Mathematica* is like working with an expert mathematical assistant. This requires a subtle change in the way you think about a problem. When working on an exercise by hand, your attention is focused on the details and quite often you can lose sight of the "big picture." *Mathematica* takes care of the details for you and frees you to focus on deciding what needs to be done next. This is not to say that the details are not important, nor does it imply that you should forgo learning how to solve the problems by hand.

Much of discrete mathematics is about understanding the relationships between objects or sets of objects and using mathematical models to capture some property of these objects. Understanding these relationships often requires that you view either the objects or the associated mathematical model in different ways.

Mathematica allows you to manipulate the mathematical models almost casually. For example, the polynomial $(x + y(x + z))^3$ can be entered in the Wolfram Language as

```
In[6]:= (x+y*(x+z))^3
Out[6]= (x+y (x+z))^3
```

The result of evaluating this expression is displayed immediately. In this case, *Mathematica* simply echoes the polynomial as no special computations were requested.

The power of having a computational tool is that a wide range of standard operations become immediately available. For example, you can expand, differentiate, and integrate just by telling *Mathematica* to do so. Suppose you decided that it would be useful to see the full expansion of the polynomial above. All you need to do is issue the appropriate command. In this case, the function you would want is the `Expand` function, which tells *Mathematica* to expand the polynomial.

```
In[7]:= Expand[%]

Out[7]= x^3+3 x^3 y+3 x^3 y^2+x^3 y^3+3 x^2 y z+6 x^2 y^2 z+3
        x^2 y^3 z+3 x y^2 z^2+3 x y^3 z^2+y^3 z^3
```

(Recall that the percent symbol, the `Out` (`%`) operator, refers to the last result.)

Perhaps you decide it would be useful to look at this as a polynomial in the variable x, with the variables y and z part of the coefficients of x. In this case, you would use the `Collect` function.

```
In[8]:= Collect[%,x]

Out[8]= x^3 (1+3 y+3 y^2+y^3)+y^3 z^3+x^2 (3 y z+6 y^2 z+3 y^3 z)+ x (3
        y^2 z^2+3 y^3 z^2)
```

To return to the factored form, simply `Factor` the previous result.

```
In[9]:= Factor[%]

Out[9]= (x+x y+y z)^3
```

We used several functions above without explanation. Rest assured that in the body of this manual we will always provide detailed explanations of the usage and syntax of new functions. The purpose of the last several paragraphs was not to introduce the commands, but to illustrate how easy it is to quickly move between different representations of the same object. Having these kinds of routine tasks performed quickly and accurately means that you are freer to experiment and explore.

A second very important benefit is that the particular computations that you choose to have *Mathematica* evaluate are performed accurately. Thus, the results you get from your experiments are much more likely to be feedback on the model you had chosen rather than nonsense arising from simple arithmetic errors.

Finally, the sheer computational power of *Mathematica* allows you to run much more extensive experiments and many more of them. This can be important when trying to establish or identify a relationship between a mathematical model and a collection of discrete objects.

It is worth making some comments about terminology and syntax. First, in this manual, we will use the terms *function* or *command* to refer to `Expand`, `Collect`, `Factor`, and so on. Wolfram Language functions will appear in blue and will be underlined to indicate that they are links to the *Mathematica* documentation.

Second, when you apply a function to one or more objects, the objects are referred to as arguments. To evaluate a function, you type its name followed by a pair of square brackets. Inside the brackets, you list the arguments, separated by commas.

```
In[10]:= Max[9,2,12,14,7,11]

Out[10]= 14
```

Even functions that do not need any arguments require the brackets. For example, the `TimeUsed` function returns the total amount of computer time that the current *Mathematica* session has used.

```
In[11]:= TimeUsed[]
Out[11]= 0.80126
```

The Basics

This section and the next are devoted to introducing you to the most essential Wolfram Language functions and concepts that will be used throughout this manual. Some of this material will be repeated, often in more depth, in the first few chapters when the topics arise naturally in conjunction with the content of the textbook. This section is focused on basic commands and the next focuses on programming.

Help

The most important command is the help command. There is extensive documentation available on all of the Wolfram Language functions, including examples of how the function is used. There are three primary ways to access this documentation. First, you can select **Wolfram Documentation** from the **Help** menu. The documentation center window will open and from there you can browse or search for the function you need.

Second, within a Notebook, you can enter a question mark (`?`) followed by the name of a function. For example, if you wanted more information on the function for computing square roots, you would enter the following:

```
In[12]:= ?Sqrt
```

Sqrt[z] or $\sqrt{z}$ gives the square root of z. ≫

A brief description is displayed. If you click on the ≫, the full documentation page will open for the function. Remember that functions discussed in this manual will appear blue and underlined. Clicking on them will open the documentation page for the function, provided you are using the *Mathematica* Notebook version of the chapter with the desktop version of *Mathematica*.

Third, the complete documentation for the most current version of the Wolfram Language is available at the website https://reference.wolfram.com.

Wolfram-Alpha Integration

Another useful feature of *Mathematica* is the integration with Wolfram-Alpha. In particular, you can use the seamless integration to take advantage of the free-form linguistic input to have Wolfram-Alpha help you determine the appropriate Wolfram Language syntax. For example, suppose you want to compute the square root of 9, but do not know about the `Sqrt` function. You invoke the free-form linguistic interpretation by beginning an input with an equal sign or selecting "Free-form input" from the new cell drop-down menu (obtained by clicking on the plus sign visible when the cursor is between cells). Then, you simply type what you want, for example, "square root of 9".

```
= square root of 9
```

When you evaluate such a cell, *Mathematica* connects to Wolfram-Alpha to interpret your input and displays the proper Wolfram Language function. *Mathematica* then evaluates that expression to obtain the result.

```
In[13]:= square root of 9
         Sqrt [9]

Out[13]= 3
```

This is a useful way to determine the right function to use. Then, you can explore the documentation to learn more about the function and its arguments in order to get precisely what you want.

Also note that beginning an input cell with two equal signs will produce results just as if you entered the query on the Wolfram-Alpha website.

Arithmetic

The Wolfram Language uses the typical notation for arithmetic. For addition and subtraction, the notation is + and – just as you would expect. The – symbol is used for negation as well. Multiplication and division are performed with * and /, and ^ is used for exponentiation.

Mathematica obeys the usual order of precedence for arithmetic operators, and parentheses serve as grouping symbols. However, brackets, braces, and angle brackets all have different meanings in the Wolfram Language and cannot be used as grouping symbols in arithmetic expressions. Thus, to compute the expression $7 + 2 \cdot \left[5 - \left(\frac{2}{3}\pi\right)^2\right]$, you would enter the following, using `Pi` for π and parentheses in place of the brackets.

```
In[14]:= 7+2*(5-(2/3*Pi)^2)

Out[14]= 7+2 (5-4 π^2/9)
```

Note that the multiplication symbol following the 2 is optional. In addition, *Mathematica* performs some algebraic simplifications automatically. If you desire additional simplification, you can use the `Simplify` function.

```
In[15]:= Simplify[%]

Out[15]= 17-8 π^2/9
```

If you prefer a floating-point approximation of the result, you can use the `N` function. This function takes one required argument, an expression, and evaluates it with floating-point arithmetic.

```
In[16]:= N[%]

Out[16]= 8.22702
```

Note that `N` can accept an optional second argument, a positive integer specifying the number of digits of precision.

```
In[17]:= N[Pi,10]

Out[17]= 3.141592654
```

Mathematica works with exact values such as integers and the symbol `Pi` differently from floating-point values. By including any decimal in an expression, *Mathematica* will treat the entire expression as a floating-point computation. Compare the following two expressions.

```
In[18]:= 2/3+3/2
Out[18]= 13/6
In[19]:= 2/3+3.0/2
Out[19]= 2.16667
```

The presence of 3.0 caused *Mathematica* to evaluate with floating-point computations rather than rational arithmetic. Note that the trailing 0 is not necessary: entering 3. is sufficient to indicate that you want the number evaluated using floating-point arithmetic.

```
In[20]:= 3./5
Out[20]= 0.6
```

This discussion illustrates the concept of a *type*. A computer can be much more efficient if it knows what kinds of things it will be working with. If the computer knows that one object is going to be a floating-point number whereas another is going to be a string, it will allocate memory differently for the two objects, for instance.

Types also allow programming languages to make use of operator overloading. This means that the symbol + means one thing when applied to two integers, something else when applied to floating-point numbers, and something completely different when applied to matrices. The concept of type is what makes it possible for *Mathematica* to figure out which version of + is called for at the time.

In the Wolfram Language, types are implemented using heads. Everything in the Wolfram Language is an expression and every expression is of the form `h[...]`, just like a function, at least in the internal representation. The symbol `h` is called the head of the expression. You can use the function `Head` to determine the head of any expression.

```
In[21]:= Head[3]
Out[21]= Integer
In[22]:= Head[3.]
Out[22]= Real
```

To reveal the internal representation of an expression, you use the `FullForm` function. Below, we see that even an expression as simple as $x + 3$ is viewed by *Mathematica* as a function applied to arguments.

```
In[23]:= FullForm[x+3]
Out[23]//FullForm=
Plus[3,x]
```

In the chapters that follow, understanding the internal representation of expressions will be very useful to us.

The functions just discussed, N, Head, and FullForm, are similar in that they each require only one argument. It is common in the Wolfram Language to apply functions such as these, particularly those whose primary effect is on the form of output, using the Postfix (//) operator, which allows you to apply a function by ending an expression with the symbol // followed by the name of the function. This is illustrated below.

```
In[24]:= E//N
Out[24]= 2.71828
In[25]:= {1,2,3}//Head
Out[25]= List
In[26]:= 3x//FullForm
Out[26]//FullForm=
Times[3,x]
```

Symbols, Assignment, and Equality

In mathematics, we talk about variables as letters that stand in for something else. In the Wolfram Language, this role is filled by symbols. The simplest definition of a symbol in the Wolfram Language is that a symbol must begin with a letter and may be followed by letters or digits. The following are all valid symbols: n, x, Pi, and a15.

Symbols can be used as a variable in an algebraic expression as in the following:

```
In[27]:= 3x^2+5x-7
Out[27]= -7+5 x+3 x^2
```

Symbols can also be used to store particular values using the assignment operator, which is called Set (=). To assign a value to a symbol, you begin with the symbol, followed by the assignment operator (the equals sign) and then the expression that you want stored in the symbol. For example, to assign the value 12 to the symbol y, you type the statement below.

```
In[28]:= y=12
Out[28]= 12
```

When a value or other expression has been assigned to a symbol, then any time that symbol appears in an expression, it is resolved to the value stored in it.

```
In[29]:= y+5x
Out[29]= 12+5 x
```

When *Mathematica* encounters an assignment statement, it first evaluates the right-hand side of the expression and then makes the assignment. You can use this fact to modify values as follows:

```
In[30]:= y=2y+1
Out[30]= 25
```

In the statement above, the right-hand side is evaluated first, meaning that the y on the right is resolved to its "old" value of 12. *Mathematica* then computes $2 \cdot 12 + 1$ and assigns the value 25 to the symbol y, overwriting the value stored earlier.

You can remove the value assigned to a symbol by using the Clear function applied to the symbol or with the Unset (=.) operator. Both of these are illustrated below.

```
In[31]:= Clear[y]

In[32]:= y=.
```

Practically any expression can be assigned to a symbol, not just numbers. For example, we can assign the algebraic expression $2y + 5x$ to the symbol f.

```
In[33]:= f=2y+5x
Out[33]= 5 x+2 y
```

Now, every time f appears, it is resolved to this expression.

```
In[34]:= Sqrt[f]
Out[34]= √(5 x+2 y)
```

Even an equation can be assigned to a symbol.

```
In[35]:= eqn=F==(9/5)*C + 32
Out[35]= F==32+9 C/5
```

Observe in the previous example the use of the equal sign as the Set (=) operator defining the symbol *eqn*. To express mathematical equality, you must use the Equal (==) relation, which consists of two equal signs. The previous statement assigns the symbol *eqn* to the mathematical equation $F = \frac{9}{5}C + 32$. Since *Mathematica* understands this to be an equation, we can, for instance, solve it. Given an equation and a symbol appearing in the equation, the Solve function solves the equation for the given symbol. We can solve for *C* as follows.

```
In[36]:= Solve[eqn,C]
Out[36]= {{C→5/9 (-32+F)}}
```

We will have more to say about the format of the output in later chapters. Here, it suffices to understand that *Mathematica* has solved the equation *eqn* and determined that C is $\frac{5}{9}(-32 + F)$.

Numbers and Strings

We have already seen that the Wolfram Language distinguishes between integers and real numbers. It also recognizes rational and complex numbers, with the symbol I used to indicate the imaginary unit.

```
In[37]:= Head[2/3]
Out[37]= Rational

In[38]:= Head[3+4I]
Out[38]= Complex
```

Strings are another basic kind of object in the Wolfram Language. You form a string by enclosing any sequence of characters within a pair of double quotes. For example, Einstein wrote,

```
In[39]:= quotation=
         "Pure mathematics is, in its way, the poetry of
            logical ideas."
Out[39]= Pure mathematics is, in its way, the poetry of logical ideas.

In[40]:= Head[quotation]

Out[40]= String
```

String may be combined with the concatenation function `StringJoin` (<>) or its operator, as demonstrated below.

```
In[41]:= quotation<>" - Einstein"

Out[41]= Pure mathematics is, in its way,
            the poetry of logical ideas. - Einstein
In[42]:= StringJoin["abc","def"]

Out[42]= abcdef
```

Lists

In the Wolfram Language, a list is an ordered sequence of expressions. Note that the elements or members of a list can be any expression whatsoever, from numbers to graphics objects. You create a list by entering the members separated by commas and enclosed in a pair of braces. For example, the following is the list of integers from 6 to 12.

```
In[43]:= L={6,7,8,9,10,11,12}

Out[43]= {6,7,8,9,10,11,12}
```

The head of a list is `List`.

```
In[44]:= Head[L]

Out[44]= List
```

The `Part` Function

Given a list, you access individual elements, and sublists, with the `Part` (`[[...]]`) function, typically using the double-bracket operator. Elements of a list are indexed beginning with 1, that is, the first element is in location 1, the second element is in position 2, etc. Thus, to access the third element of the list `L`, you enter the following.

```
In[45]:= L[[3]]

Out[45]= 8
```

Negative integers can be used with `Part ([[...]])` in order to refer to elements of a list counting from the end. That is, −1 refers to the last element of the list, −2 refers to the next to last, etc.

```
In[46]:= L[[-2]]
Out[46]= 11
```

The Wolfram Language also provides two functions, `First` and `Last`, that can be used to access those elements. These have the same effect as calling `Part ([[...]])` with index 1 or −1, but are more descriptive.

```
In[47]:= First[L]
Out[47]= 6
In[48]:= Last[L]
Out[48]= 12
```

Lists can be nested, as in the example below.

```
In[49]:= nestedL={{1,2,3},{4,5,6},{7,8,9}}
Out[49]= {{1,2,3},{4,5,6},{7,8,9}}
```

Using `Part ([[...]])` with a single index will refer to the corresponding sublist.

```
In[50]:= nestedL[[2]]
Out[50]= {4,5,6}
```

To obtain individual elements in the sublist, you can either apply `Part ([[...]])` a second time or follow the index of the sublist with a comma and an index into the sublist. Both of the following access the first element of the second sublist.

```
In[51]:= nestedL[[2]][[1]]
Out[51]= 4
In[52]:= nestedL[[2,1]]
Out[52]= 4
```

`Part ([[...]])` is also used to obtain sublists. To do this, you provide a list (enclosed in braces) of the desired indices to `Part ([[...]])`. Note that the order of the indices determines the order of the output. For example, to obtain the sublist of L consisting of the third, seventh, and fifth elements, you enter the following.

```
In[53]:= L[[{3,7,5}]]
Out[53]= {8,12,10}
```

You can use the `Span` (`;;`) operator in conjunction with `Part` (`[[...]]`) in order to obtain a sublist. Within `Part` (`[[...]]`), ***a***;;***b*** refers to the sublist of elements from index ***a*** to index ***b***. The following produces the sublist of `L` from index 2 through 5.

```
In[54]:= L[[2;;5]]

Out[54]= {7,8,9,10}
```

You can use 1 and −1 within a `Span` (`;;`) to refer to the beginning and end of the original list. You can also simply omit ***a*** or ***b*** and *Mathematica* will interpret that `Span` (`;;`) as beginning at the start or stopping at the end of the list, respectively.

```
In[55]:= L[[;;3]]

Out[55]= {6,7,8}

In[56]:= L[[5;;]]

Out[56]= {10,11,12}
```

We mentioned above that every expression in the Wolfram Language is, internally, represented as a head applied to a number of arguments. Lists are no different, as `FullForm` reveals.

```
In[57]:= FullForm[L]

Out[57]//FullForm=
         List[6,7,8,9,10,11,12]
```

This recognition leads us to two observations.

First, `Part` (`[[...]]`) is not a function that applies exclusively to lists. Rather, it can be used with any expression, with the index referring to the positions of the elements within the brackets. For example, consider the sum below.

```
In[58]:= sum=a+b+c+d+e

Out[58]= a+b+c+d+e

In[59]:= FullForm[sum]

Out[59]//FullForm=
         Plus[a,b,c,d,e]
```

We can use `Part` (`[[...]]`) to access the individual elements being summed.

```
In[60]:= sum[[3]]

Out[60]= c
```

The second observation is that this universality of `Part` (`[[...]]`) provides a natural meaning to the index 0. Namely, index 0 refers to the head of the expression.

```
In[61]:= L[[0]]

Out[61]= List
```

```
In[62]:= sum[[0]]

Out[62]= Plus
```

The `Range` *Function*

Having discussed `Part` (`[[...]]`) for accessing elements of lists (and other expressions), we now turn to two important functions for creating lists. The first of these is `Range`, which is used to create simple lists comprised of sequential numbers.

`Range` can accept one, two, or three arguments. Given a single argument, ***max***, `Range` produces the list of positive integers beginning with 1 and up to ***max***. For example, the code below creates the list of integers from 1 to 10.

```
In[63]:= Range[10]

Out[63]= {1,2,3,4,5,6,7,8,9,10}
```

With two arguments, ***min*** and ***max***, the output of `Range` is the numbers beginning with ***min*** and up to ***max***. For example, the following produces the list of integers from −3 to 7.

```
In[64]:= Range[-3,7]

Out[64]= {-3,-2,-1,0,1,2,3,4,5,6,7}
```

Adding a third argument, ***step***, `Range` will output the list beginning at ***min*** up to a maximum of ***max*** and increasing by ***step*** each time. For example, to produce the even integers from 10 to 20, you would give a ***step*** of 2.

```
In[65]:= Range[10,20,2]

Out[65]= {10,12,14,16,18,20}
```

Note that ***max*** is not necessarily an element of the output, it serves as an upper bound. In the example below, the maximum will not be included in the list, given the ***step***.

```
In[66]:= Range[1,10,4]

Out[66]= {1,5,9}
```

Also note that the arguments to the `Range` function are not required to be integers, as the following illustrate:

```
In[67]:= Range[3.7]

Out[67]= {1,2,3}

In[68]:= Range[2.3,7.9]

Out[68]= {2.3,3.3,4.3,5.3,6.3,7.3}

In[69]:= Range[.1,1.5,.2]

Out[69]= {0.1,0.3,0.5,0.7,0.9,1.1,1.3,1.5}
```

The `Table` function

The `Table` function is a more flexible way to create lists. `Table` requires two arguments. The first argument is an expression, usually written in terms of a variable called the table variable or table index. The second argument specifies the values that the table variable are to take. The result of `Table` is that the expression given as the first argument is evaluated for each of the specified values of the table index, and a list is built out of those results.

To make this more precise, consider the example below, which produces the list of the squares of the first ten positive integers.

```
In[70]:= Table[i^2,{i,10}]
Out[70]= {1,4,9,16,25,36,49,64,81,100}
```

In the above, the symbol `i` is used as the table variable. The first argument to `Table`, the expression `i^2`, indicates that the list that is produced will contain the squares of the values of `i`. The second argument to `Table`, called the iteration specification, has a variety of forms. In the above, the first element of the iteration specification identifies `i` as the table variable. The second element, 10, indicates that the variable `i` will be assigned the integers from 1 up to a maximum of 10. Note the similarity to the syntax of `Range`.

There are two more forms of the iteration specification corresponding to the other ways to invoke `Range`. To specify both a minimum and maximum for the table variable, you give the iteration specification as {*var*, *min*, *max*}, where *var* is the table variable. The following produces the squares of the integers from 5 to 12.

```
In[71]:= Table[i^2,{i,5,12}]
Out[71]= {25,36,49,64,81,100,121,144}
```

A *step* is specified by the iteration specification {*var*, *min*, *max*, *step*}. The following outputs the list of squares of the first 10 even integers.

```
In[72]:= Table[i^2,{i,2,20,2}]
Out[72]= {4,16,36,64,100,144,196,256,324,400}
```

You can also identify a specific *list* of possible values by giving the iteration specification {*var*, *list*}. For example, the following produces the list of the squares of the first 6 primes.

```
In[73]:= Table[i^2,{i,{2,3,5,7,11,13}}]
Out[73]= {4,9,25,49,121,169}
```

Note that this form can be used with nonnumeric iteration specifications. For example, the following uses the `StringJoin` (<>) operator with `Table` to create a list of the five English words of the form "p"-vowel-"t".

```
In[74]:= Table["p"<>v<>"t",{v,{"a","e","i","o","u"}}]
Out[74]= {pat,pet,pit,pot,put}
```

The last possible iteration specification consists of a single integer, omitting the table variable. The output from `Table` will be that number of copies of the first argument. For example, the following produces a list of 10 zeros.

```
In[75]:= Table[0,10]
Out[75]= {0,0,0,0,0,0,0,0,0,0}
```

Note that the same effect can be obtained with `ConstantArray`.

```
In[76]:= ConstantArray[0,10]
Out[76]= {0,0,0,0,0,0,0,0,0,0}
```

The table below summarizes the possible forms of the iteration specification for `Table`.

count	*count* copies
{*i*, *max*}	*i* ranges from 1 to *max*
{*i*, *min*, *max*}	*i*ranges from *min* to *max*
{*i*, *min*, *max*, *step*}	*i* ranges from *min* to *max* by *step*
{*i*, *list*}	*i* ranges over elements of *list*

The `Map` Function

The `Map` (`/@`) function is used to apply a function to a list of elements. As a function, `Map` (`/@`) takes two arguments: the name of a function and a list. (Technically, the second argument can be any expression, not just a list, but we will not explore that here.) For example, the following applies the square root function to a list of elements.

```
In[77]:= Map[Sqrt,{4,9,16,25,36}]
Out[77]= {2,3,4,5,6}
```

Observe that the function used in `Map` (`/@`) is the name of a function, not an expression, and there is no variable involved. `Map` (`/@`) should be viewed as a more fundamental version of `Table`, and in fact, `Table` could be defined as a particular application of `Map` (`/@`).

Like many of the fundamental commands, `Map` (`/@`) has an operator form, which is illustrated below.

```
In[78]:= Sqrt/@{4,9,16,25,36}
Out[78]= {2,3,4,5,6}
```

It can be helpful to look at the result of applying `Map` (`/@`) with an undefined symbol in place of the function.

```
In[79]:= function/@Range[5]
Out[79]= {function[1],function[2],function[3],function[4],function[5]}
```

This reveals the operation of `Map` (`/@`) very clearly. When *Mathematica* applies `Map` (`/@`) to a symbol and a list, it applies the symbol to each member of the list. If the symbol is a defined function, *Mathematica* would then evaluate the function at those values.

The `Apply` Function

`Apply` (`@@`) is another essential function. Keeping in mind that every expression in the Wolfram Language consists of a head followed by a list of arguments in brackets, `Apply` (`@@`) simply replaces the head of an expression.

We can make this explicit by using `Apply` (`@@`) with two undefined heads. First, we define an expression `applyExample`.

```
In[80]:= applyExample=head1[1,2,3,4,5]

Out[80]= head1[1,2,3,4,5]
```

Since `head1` is undefined, *Mathematica* simply echoes the definition. Using `Apply` (`@@`), we can replace the head `head1` with a new head, say `head2`.

```
In[81]:= head2@@applyExample

Out[81]= head2[1,2,3,4,5]
```

One important use of `Apply` (`@@`) is to apply a function to a list. More precisely, given a list and a function, `Apply` (`@@`) can be used to evaluate the function with the elements of the list as arguments. For example, using `Apply` (`@@`) and the `Plus` function, we can sum the elements of a list.

```
In[82]:= Plus@@{1,2,3,4,5}

Out[82]= 15
```

Using symbols instead of actual numbers and `FullForm` reveals that, just as before, `Apply` (`@@`) is causing the head of the expression to be replaced, this time `List` is replaced by `Plus`.

```
In[83]:= applyExample2={val1,val2,val3,val4,val5}

Out[83]= {val1,val2,val3,val4,val5}

In[84]:= FullForm[applyExample2]

Out[84]//FullForm=
         List[val1,val2,val3,val4,val5]

In[85]:= Plus@@applyExample2

Out[85]= val1+val2+val3+val4+val5

In[86]:= FullForm[%]

Out[86]//FullForm=
         Plus[val1,val2,val3,val4,val5]
```

Printing

We end this section with a brief description of the `Print` function. In most cases, the only information we need to display is the final result of some computation. However, occasionally we may want to explicitly cause some information to be displayed. For example, in troubleshooting functions you create, it is common to have information printed as the function is evaluated so as to track internal variables.

The `Print` function can be applied to any number of arguments. The result is that the values of the arguments are displayed together on a single line of output. For example, the following displays the value of 2 + 3, the string "hello", and the list `L` from above.

```
In[87]:= Print[2+3,"hello",L]

         5hello{6,7,8,9,10,11,12}
```

Observe that the arguments to `Print` are evaluated, but no space is added. Also observe that `Print` does not produce output (technically, its output is the special symbol `Null`).

Similar to `Print` is `Row`, which is applied to a list and displays the elements in a row.

```
In[88]:= Row[{2+3,"hello",L}]

Out[88]= 5hello{6,7,8,9,10,11,12}
```

`Row` can take a second argument, which is used as a separator between elements of the list. For example, to add a comma and space between the elements when displayed, enter the following.

```
In[89]:= Row[{2+3,"hello",L},", "]

Out[89]= 5, hello, {6,7,8,9,10,11,12}
```

Note that both `Print` and `Row` will split lines as needed to fit to the width of your window.

Finally, `Column` is used to print the elements of a list vertically.

```
In[90]:= Column[{2+3,"hello",L}]

Out[90]= 5
         hello
         {6,7,8,9,10,11,12}
```

`Column` can accept a second argument specifying the alignment of the elements. Valid options are `Left`, `Center`, and `Right`.

```
In[91]:= Column[{2+3,"hello",L},Center]

Out[91]=       5
             hello
         6,7,8,9,10,11,12
```

A third argument can be used to increase the vertical spacing between rows.

```
In[92]:= Column[{2+3,"hello",L},Center,1]


Out[92]=       5

             hello

         6,7,8,9,10,11,12
```

Programming Preliminaries

This section is intended for those readers who have little or no previous exposure to programming. We will endeavor to provide you with enough information to get you started so that you can work productively with the Wolfram Language. For further information, you are encouraged to consult the Wolfram Language guides and tutorials, which will provide you with additional examples of the use of *Mathematica*'s programming facilities.

All programming languages provide a few basic means for the construction of algorithms. On the most basic level, a computer program is a sequence of instructions that the computer executes one after the other. Programs become more sophisticated when you start changing the flow of execution. The Wolfram Language provides the same sort of mechanisms for flow control as is found in traditional programming languages such as C. While the syntax varies from one computer language to the next, there are two primary kinds of control structures used: branching and iteration.

The Wolfram Language, from a programming perspective, is a multi-paradigm language, which gives it flexibility. However, given the style of syntax, it is most natural to work with it as a functional language, which gives it a substantially different flavor from an imperative, or procedural, paradigm. Those readers with experience with imperative languages, and object-oriented languages particularly, will find that a functional approach requires a shift in the way you think about programs.

Branching

We will first discuss the concept of branching and its implementation in the Wolfram Language. Branching is a mechanism that allows you to choose between expressions based on conditions that can only be determined during a program's execution or evaluation. This is also called a selection or conditional statement.

If

As an example, suppose that you want to display a message based on whether a particular value is positive. First, we assign a value to the symbol `z`.

```
In[93]:= z=5

Out[93]= 5
```

The following will output the string "That's positive" if the value stored in `z` is greater than zero.

```
In[94]:= If[z>0,
            "That's positive"
          ]

Out[94]= That's positive
```

First note that, in order to begin a new line within an input cell, you simply press the return or enter key on the alpha-numeric keyboard (without holding the shift key down). The line breaks and extra spaces are not required, however, and in fact *Mathematica* ignores them entirely. That said, they often make functions easier to read and understand.

Typically, the condition in an `If` depends on a value input to a function, some intermediary calculation, or a value that changes during execution. In these examples, think about the symbol `z` as storing some value that varies based on some other computations. For instance, `z` could be the value of some function at a particular point. In that case, the value of `z` would depend on which point was chosen.

Let us now dissect the expression above. You can think of the expression as an application of a function `If` to two arguments. It perhaps seems odd to think of `If` as a function, especially if you are familiar with imperative languages, and it may be more comfortable to think of the input above merely as an expression with head `If`. However, `If` is a perfectly valid function, in that it accepts arguments and returns output.

The first argument to `If` is a conditional expression, that is, an expression that *Mathematica* can evaluate to `True` or `False`. Conditional expressions may include expressions that include relational operators

(e.g., <, <=, ==, >, >=, =!), logical operators (&&, ||, !), or functions that return logical values (True or False). We will see many examples of conditional expressions in Chapter 1.

The second argument to If is the expression evaluated in the case that the condition evaluates to True. This is commonly referred to as the "then clause" in many languages. Note that you can include more than one expression in the "then clause" by separating them with semicolons.

When you evaluate the expression above, *Mathematica* first evaluates the conditional expression $z > 0$. Since this is a true statement (because z happened to be 5), *Mathematica* evaluates the second argument of the If. The result of evaluating the "then clause," that is, the second argument, is the output from the If expression. If the first argument had not evaluated to True, then the second argument would not have been evaluated and the output of the function would have been Null.

Below is another example, in which the conditional statement is false.

```
In[95]:= If[z≥10,
           "That has at least two digits."
         ]
```

Notice that, in this case, nothing is displayed. In fact, the output is the symbol Null, which we can see by applying FullForm to the outcome of the expression.

```
In[96]:= FullForm[%]

Out[96]//FullForm=
         Null
```

The Else Clause

Often, you will want to take one action if a condition is true and a different action if a condition is false. The optional third argument to If, called the "else clause" in many languages, allows you to specify an expression to be evaluated if the condition is false.

```
In[97]:= If[z<0,
           "Negative",
           "Not negative"
         ]

Out[97]= Not negative
```

In the expression above, *Mathematica* determines that the result of evaluating $z < 0$ is False. Since there is a third argument to the If expression, *Mathematica* then evaluates that third expression causing it to be the output for the cell.

Note that, in the Wolfram Language, many expressions are neither True nor False. For example, the expression in the first argument of an If could evaluate to a number, as below.

```
In[98]:= If[z+2,
           "true",
           "false"
         ]

Out[98]= If[7,true,false]
```

To handle cases such as this, `If` allows for a fourth argument to be evaluated whenever the first argument resolves to any value other than `True` or `False`.

```
In[99]:= If[z+2,
            "true",
            "false",
            "neither"
          ]
Out[99]= neither
```

This is particularly useful to ensure that the expressions you create are robust, that is, they can handle "bad data." For example, if the symbol `z` were assigned to a letter, *Mathematica* will not evaluate a comparison using `Greater` (>).

```
In[100]:= z2="x"
Out[100]= x
In[101]:= z2>0
Out[101]= x>0
```

In this case, the `If` expression above will simply be echoed.

```
In[102]:= If[z2<0,
            "Negative",
            "Not negative"
          ]
Out[102]= If[x<0,Negative,Not negative]
```

The fourth argument can be used to call attention to the bad value.

```
In[103]:= If[z2<0,
            "Negative",
            "Not negative",
            "Something's wrong"
          ]
Out[103]= Something's wrong
```

Which

Many programming languages provide an "else if" structure. In the Wolfram Language, this is accomplished via the `Which` function.

The `Which` function requires an even number of arguments in test/value pairs. Consider the following example.

```
In[104]:= z3=1
Out[104]= 1
```

In[105]:=
```
Which[
  z3==0,
  "z3 is 0",
  z3==1,
  "z3 is 1",
  z3==2,
  "z3 is 2"
]
```

Out[105]=
```
z3 is 1
```

The `Which` expression above has six arguments, that is, three test/result pairs. The first argument, `z3==0`, the third argument, `z3==1`, and the fifth argument, `z3==2`, are expressions that test the value of `z3` against the integers 0, 1, and 2. The result arguments, that is, the second, fourth, and sixth, express the output should the corresponding test evaluate to `True`. In the above, since `z3` was set to 1, the output is the value expression following the test comparing `z3` to 1.

When *Mathematica* encounters a `Which` expression, it first evaluates the first argument. If the result of the first argument is `True`, then it evaluates the second argument and that result is the output of the `Which`. If the first argument does not evaluate to `True`, then *Mathematica* moves on to the third argument and evaluates it. If the third argument evaluates to `True`, then the fourth argument is evaluated and is the output. Continuing in the same way, *Mathematica* evaluates each odd argument in turn until one produces `True`, at which point the next result argument is evaluated and that value is the output of the `Which`. If no test argument evaluates to `True`, then the output of the `Which` will be `Null`.

Note that once a test expression evaluates to `True`, *Mathematica* evaluates the corresponding value expression and then terminates evaluation of the `Which`. In particular, no further test expressions are evaluated, so if multiple test expressions evaluate to `True`, the output is determined by the value expression from the first such test.

To illustrate this, we modify the `Which` expression above to test for less than or equal. We also use the fact that each argument can be given as a sequence of expressions separated by semicolons in order to embed `Print` statements within the test arguments. This allows us to see exactly which arguments in the `Which` are being evaluated.

In[106]:=
```
Which[
  Print["testing <=0"];z3≤0,
  "z3 is at most 0",
  Print["testing <=1"];z3≤1,
  "z3 is at most 1",
  Print["testing <=2"];z3≤2,
  "z3 is at most 2"
]
```

```
testing <=0

testing <=1
```

Out[106]=
```
z3 is at most 1
```

Observe that "testing <=2" is never printed, indicating that the third test argument is never evaluated. Once a true test is encountered, any further arguments are ignored.

To include an "else clause" within a `Which`, that is, to specify output for the case where none of the tests evaluate to `True`, simply include a final test/result pair with the test expression set to `True`. If any of the other test expressions are satisfied, then *Mathematica* will never encounter the final test. However, if none of the other tests are true, then `True` certainly is satisfied and the final result will be evaluated. This is illustrated below.

```
In[107]:= Which[
            z3==5,
            "z3 is 5",
            z3==6,
            "z3 is 6",
            True,
            "z3 is neither 5 nor 6"
          ]
Out[107]= z3 is neither 5 nor 6
```

Iteration

The previous subsection showed how to use branching in the Wolfram Language to evaluate different expressions depending on whether or not a specified condition was met. In this section, we look at ways to repeat a block of code. Iteration is the mechanism for doing a given task repeatedly and is typically accomplished by a loop structure.

For Loops

One of the most commonly used types of iteration is the `For` loop. The simplest kind of `For` loop executes a statement for each integer in a particular range. The example below prints the squares of the integers from 3 to 5.

```
In[108]:= For[i=3,
            i≤5,
            i++,
            Print[i^2]
          ]
          9
          16
          25
```

A `For` expression in the Wolfram Language has four arguments with the following structure: For[*init*, *test*, *incr*, *body*]. The first argument, *init*, is the initialization statement. This is typically, such as `i=3`, an assignment of a symbol, called the loop variable, to an initial value. Note that the loop variable in a `For` loop is not automatically local to the loop, so if `i` had meaning before the loop above was evaluated, that value is now changed. Later in this chapter, we will see how to define scope for variables.

The second argument to a `For` loop, the *test*, defines the bound of the loop. This is commonly an inequality. In the example, `i<=5` indicates that the loop will continue until the value of the loop variable is greater than 5.

The third argument, ***incr***, is the increment. This argument defines how the loop variable is to be modified each time the body of the loop completes. Be certain that your increment expression modifies the loop variable, or the loop will never terminate. In the example, we used the `Increment` (++) operator, which is equivalent to `i=i+1`. The `Decrement` (--) operator, which decreases the value of the variable by 1, is also popular for this purpose, but any expression that modifies the value of the loop variable is valid for the third argument, provided that repeated execution of ***incr*** will eventually cause the ***test*** to fail.

The final argument is the ***body*** of the loop. This argument contains the expression, or sequence of expressions separated by semicolons, to be executed.

Now that we have established the meaning of each of the arguments to `For`, let us trace through what *Mathematica* actually does when evaluating a loop like the above.

First, *Mathematica* evaluates the initialization expression. In our example, this sets the loop variable `i` to 3.

Second, *Mathematica* immediately evaluates the ***test***. If ***test*** evaluates to `False`, then the loop is terminated. Observe that this makes `For` loops that never execute their body possible, if the initialization assignment causes the ***test*** to immediately fail. This is a sometimes useful technique.

Third, assuming the initial value passes the ***test***, the ***body*** is evaluated next.

Each time the body has been evaluated, evaluation then moves to the third argument, the increment. Again, the increment argument needs to modify the loop variable in such a way as to eventually force the test to fail. Otherwise, you will create an infinite loop, which can cause *Mathematica* to crash and you to lose your work. Always be careful with loops and save your work before evaluating them.

After each increment, ***test*** is evaluated. If the ***test*** is `True`, then flow returns to the ***body***, otherwise the loop is terminated.

Note that the output of a `For` loop is always `Null`. They are typically used not to produce output of their own, but to repeatedly perform some action that affects values or structures stored in symbols.

`While` Loops

A `While` loop is a more general type of loop structure. A `While` expression in the Wolfram Language involves only two arguments: a test and a body. In a `While` loop, the test and body are evaluated alternately until the test fails to produce `True`.

Consider the following example.

```
In[109]:= i=2;
          While[i<10^7,
            Print[i];
            i=i^2+2
          ]

          2

          6

          38

          1446

          2090918
```

Let us look at that example carefully. First, we assigned the value 2 to the symbol `i`. This is similar to the initialization in a `For` loop, but must take place outside the structure of the `While` expression. This should be viewed as adding flexibility, since in many cases the initialization process itself can be quite involved. The first argument is the conditional statement that controls the loop. This can be any condition you want. The condition is followed by the body of the loop. The statement sequence is executed repeatedly until the condition is false.

In our example, there are two expressions in the body argument. First, the current value of `i` is printed. Then, the value of `i` is changed to the result of squaring it and adding 2. This continues as long as the value of `i` is less than 10^7.

It is very important in a `While` loop to be sure that the body of the loop will eventually have the effect of making the controlling condition false. Think about what would happen if the second expression in the body of the previous loop had been `i=i-2`. In that case, `i` would have started out equal to 2. It would then become 0, then −2, then −4, then −6, etc, and it would never exceed 10^7, so it would never cease—an infinite loop. It requires great care to avoid creating infinite loops in `While` loops, even more so than with `For` loops.

Do Loops

A third kind of loop in the Wolfram Language is the `Do` loop. While the `Do` loop is less flexible than `While`, its syntax and unique semantics make it very useful.

A `Do` expression requires two arguments, but unlike both `While` and `For`, the body of the loop is the first argument. The second argument defines a loop variable and specifies its iteration using the same syntax as the `Table` function. In fact, `Do` and `Table` operate in very similar ways, the difference being in the output. Where `Table` builds a list from the results of evaluating its body, `Do` should be thought of as merely executing the commands in its body and outputs `Null`, just as `For` and `While`.

For reference, we repeat the table of allowed iteration specifications from above.

count	*count* copies
{*i*, *max*}	*i* ranges from 1 to *max*
{*i*, *min*, *max*}	*i*ranges from *min* to *max*
{*i*, *min*, *max*, *step*}	*i* ranges from *min* to *max* by *step*
{*i*, *list*}	*i* ranges over elements of *list*

Below, we use a `Do` loop to print the squares of a list of integers.

In[111]:=
```
Do[
  Print["The square of ",i," is ",i^2],
  {i,{2,5,6,11,8}}
]
```

```
The square of 2 is 4

The square of 5 is 25

The square of 6 is 36

The square of 11 is 121

The square of 8 is 64
```

Premature Loop Exit

Sometimes it is necessary to terminate a loop prematurely. This may be in order to prevent an error or because the logic of a particular problem dictates that it must. In imperative and procedural languages, it is common to use "break" statements to terminate a loop or "return" statements to terminate a loop with a particular value. The Wolfram Language includes both of these.

`Break` will immediate exit the enclosing `Do`, `For`, or `While` loop. Note that `Break` must be called with brackets but no argument.

Similarly, `Return` will also terminate a loop in which it is contained. `Return` can also be called with no argument, but it is typically given an argument, in which case that argument will be the output from the loop it is terminating. `Return` can also be used within a function definition or module (described below) to prematurely terminate evaluation of those structures.

In this manual, we generally avoid the use of both `Break` and `Return`. Part of the reason for this is practical. In particular, `Return` behaves somewhat differently in the Wolfram Language than in many other programming languages, specifically with regard to its scope. While its behavior in the Wolfram Language is completely predictable, it can be confusing if you are used to procedural languages. The other part of the reason for avoiding `Break` and `Return` is stylistic. The Wolfram Language has a functional flavor, whereas `Break` and `Return` are more properly part of imperative languages.

Instead, we will use the `Catch` and `Throw` mechanism. In some languages, "catch" and "throw" are used for error handling. In the Wolfram Language, they are used for more general flow control and short circuiting of loops. They have a more functional style than `Break` and `Return` and, unlike `Return`, their scope is explicit.

The basic idea of `Catch` and `Throw` is that you surround an expression, such as a loop construct or sequence of expressions, within `Catch`. The first time `Throw` is encountered, evaluation within the `Catch` is terminated and the output of the `Catch` block is set to be the argument of `Throw`.

Consider the example below, which determines the smallest positive integer n for which the total stopping time of n is greater than 50. (The reader is encouraged to research the Collatz conjecture, which forms the basis of this example.)

```
In[112]:= n=1;
          Catch[
            While[True,
              n=n+1;
              m=n;
              count=0;
              While[m≠1,
                If[OddQ[m],
                  m=3*m+1,
                  m=m/2
                ];
                count=count+1;
```

```
            If[count>50,
              Throw[n]
            ]
          ]
        ]
      ]
```

Out[113]= 27

In the above, the first `While` loop is intentionally an infinite loop—the condition under which it continues is the expression `True`. The inner `While` loop executes the Collatz process, which, given an integer m, either multiples by 3 and adds 1 if it is odd or divides by 2 if it is even. Once this is done, `count` is incremented. The inner `While` loop is conditioned on the relation $m \neq 1$, so the loop will continue until m becomes 1. When the inner `While` loop terminates, the value of count will be the number of steps of the Collatz process required to take the integer n to 1. This value is called the total stopping time of n.

The `If` expression within the inner `While` watches for the value of `count` to exceed 50. When `count` is found to be greater than 50, the `Throw` is encountered with argument `n`. This causes evaluation to stop, terminating both loops, and causing the value of `n` to be the outcome of the `Catch`.

Defining Functions and Modules

In the Wolfram Language, functions and programs are nearly synonymous. Here, we will explain how to define functions in the Wolfram Language and how the module structure is used to protect symbol definitions.

A function definition consists of four elements: the symbol used to name the function, the definition of the allowed arguments, the assignment operator, and the body of the function.

We first comment on the assignment operator. Thus far, we have used `Set` (=) to define values for symbols. When defining a function, however, you should always use `SetDelayed` (:=). This prevents the body of the function from being prematurely evaluated, which can result in a variety of unexpected behavior.

Consider the simple function definition below, which creates a function that simply adds its arguments.

In[114]:=
```
mySum[a_,b_]:=a+b
```

To the left of the `SetDelayed` (:=) operator is the name of the function we are defining, `mySum`, followed by square brackets surrounding the argument specification. The arguments are given as a list of patterns. We will return to patterns momentarily. On the right side of the assignment is the expression defining the function, written in terms of the names of the arguments.

Observe the function behaves as expected.

In[115]:=
```
mySum[1,2]
```

Out[115]= 3

Now, we briefly explain the patterns that define the parameters to the function. In the Wolfram Language, whenever you attempt to evaluate an expression, it looks to see whether the head of the expression is

known, and, if so, whether the arguments match the pattern of the definition associated with the head. If we attempted to call `mySum` with anything other than two arguments, *Mathematica* would simply echo the input, indicating that it failed to find a definition for that form of an expression.

```
In[116]:= mySum[1,2,3]

Out[116]= mySum[1,2,3]
```

At the heart of any pattern is the `Blank` (_), which is entered as an underscore. A `Blank` (_) is the generic wild card in the Wolfram Language, matching any single expression. For example, `mySum` is perfectly suited to adding two polynomials since a polynomial is an expression.

```
In[117]:= mySum[3x^2+5x+7,2x-3]

Out[117]= 4+7 x+3 x^2
```

By preceding a `Blank` (_) with a symbol, such as `x` or `L`, you effectively name the expression that matches that particular `Blank` (_). In our example, the symbols `a` and `b` are identified with the arguments and then used in the function definition.

You can restrict the types of expressions that are matched by a `Blank` (_) by following the underscore with the name of a head. For example, to restrict the function to apply only to integers, which have head `Integer`, you would define it as follows.

```
In[118]:= integerSum[a_Integer,b_Integer]:=a+b
```

This function works just as before on integers, but will not be applied if other kinds of expressions are entered.

```
In[119]:= integerSum[1,2]

Out[119]= 3

In[120]:= integerSum[3x^2+5x+7,2x-3]

Out[120]= integerSum[7+5 x+3 x^2,-3+2 x]
```

Note that symbols used to name patterns are automatically local to the function definition. That is, they do not retain their value once the function is complete. Below, we see that `a` can be assigned a value and that value neither affects the evaluation of `mySum` nor is modified by it.

```
In[121]:= a=23

Out[121]= 23

In[122]:= mySum[1,2]

Out[122]= 3

In[123]:= a

Out[123]= 23
```

This is not true, however, for other variables you may introduce in the body of the function. For example, the function below, given a positive integer, uses the `StringJoin` (<>) operator and a `Do` loop to form a string consisting of the given number of "x"s. (Note that there are certainly better ways to achieve this result, but this example illustrates the relevant concept.)

```
In[124]:= xstring[n_]:=
            (string="";
             Do[string=string<>"x",{n}];
             string)
```

Note that the parentheses are necessary to prevent the first semicolon from terminating the assignment. Also note that the output of a semicolon separated sequence of expressions is the value of the final expression. Since `Do` has output `Null`, we must issue the expression string in order to obtain the desired output.

```
In[125]:= xstring[5]

Out[125]= xxxxx
```

Unlike `mySum` and the parameters `a` and `b`, the value of `string` remains after the function has been executed.

```
In[126]:= string

Out[126]= xxxxx
```

More concerning, if `string` had previously been assigned a value, it would have been replaced.

The Wolfram Language provides the `Module` structure to encapsulate program definitions and, in particular, define variables to have local scope. We illustrate with a second example.

```
In[127]:= ystring[n_]:=Module[{string,i},
            string="";
            For[i=1,i≤n,i++,
              string=string<>"y";
            ];
            string
          ]
```

The `Module` encloses the entire body of the function. It begins with a list of the variables to be used that are local to the function. This list is followed by the semicolon separated sequence of expressions comprising the function body. The output of `Module` is the last expression evaluated, so again we must end with the expression `string`.

Note that both of the variables used in `ystring` have values: `string` from its use in `xstring` and `i` from the last time it was used in a loop.

```
In[128]:= string

Out[128]= xxxxx

In[129]:= i

Out[129]= 4371938082726
```

If we call *ystring* on a positive integer, it produces the expected result.

```
In[130]:= ystring[11]

Out[130]= yyyyyyyyyyy
```

However, `string` and `i` are not changed.

```
In[131]:= string

Out[131]= xxxxx

In[132]:= i

Out[132]= 4371938082726
```

It is good practice to use `Module` when defining any but the simplest functions.

Mathematica Versions

This manual was created using *Mathematica 11*. Most of the functions used, however, are compatible with older versions of the software.

On-Line Material

The files for this manual, including both the PDF and *Mathematica* Notebook versions of all chapters, are available at the website for the eighth edition of *Discrete Mathematics and Its Applications* by Kenneth Rosen: www.mhhe.com/rosen. This site includes many other kinds of supplementary material for students and instructors.

In addition to the chapters of the manual, the website also includes *Mathematica* packages containing the useful functions defined in the chapters that you may use to further explore the concepts of discrete mathematics. To use these packages, you need to download them to your computer and load them into your *Mathematica* session. The packages are saved as files named "Chapter##.wl", where "##" refers to the two-digit chapter number.

To load a package, you use the `Get` (<<) operator. Suppose that Chapter01.wl is located in the directory /Users/myaccount/DiscreteMath/ on your computer or in your account for one of the cloud-based Wolfram systems. Then, you would execute the following expression, replacing the directory shown with the directory on your computer.

```
<<"/Users/myaccount/DiscreteMath/Chapter01.wl"
```

If you have saved the notebook you are working with, you can determine its directory with `NotebookDirectory[]`. This can be used in conjunction with `StringJoin` (<>) to load a file in the same directory as your Notebook.

```
<<(NotebookDirectory[]<>"Chapter00.wl")
```

Loading the package will give you access to the functions defined in that chapter, without needing to open and evaluate the function definitions in that chapter's Notebook file.

Exercises

1. Compute $17 \cdot \left(126^{34} - 93\right)$.
2. Form the `List` of the first 100 positive integers that are 3 greater than a multiple of 7, using the `Table` function.
3. Use a `For` loop to `Print` the string "Hello World!" 10 times.
4. Use a `While` loop to `Print` the string "Hello World!" 10 times.
5. Use a `Do` loop to `Print` the string "Hello World!" 10 times.
6. `EvenQ` applied to an integer returns `True` if the argument is even. Loop over the list you created in Exercise 2, and within the loop, use an `If` expression to `Print` "even" or "odd" for each element of the list.
7. Define a function `f` by the formula $f(x) = x^2 + 3x - 2$ and then use `Map` to apply that function to the list $\{-5, -4.5, -4, \ldots, 4, 4.5, 5\}$.
8. Write a function using `Module` that has one parameter with head `List` and outputs the list in reverse order. You should use only functions discussed in this Introduction.