CAMBRIDGE

Cambridge University Press
978-1-107-11666-5 - Essentials of Programming in Mathematica®
Paul Wellin
Excerpt
More information

I

Programming with *Mathematica*

Introduction to programming · *Your first Mathematica program* · *Programming paradigms* · *Creating programs* · *Getting started* · *Starting and running Mathematica* · *Mathematical expressions* · *Functions* · *Lists* · *Semicolons* · *Alternative input syntax* · *Comments* · *Getting help* · *Errors* · *Getting out of trouble* · *Function information* · *Documentation* · *Notes and further reading*

Mathematica is a large system used across an astonishing array of disciplines – physics, bioinformatics, geo-science, linguistics, network analysis, optics, risk management, software engineering, and many more. It integrates tools for importing, analyzing, simulating, visualizing, reporting, and connecting to other programs. Underlying all of these tools is a modern programming language with which you can extend the things you can do with *Mathematica* almost limitlessly.

This book focuses on the programming language, but before we dive into details of syntax and structure, it will be helpful to look at some of the essential elements of actual programs to give you a better sense of what it means to program with *Mathematica*. So, to start, let us walk through the creation of a short program so you can see how a *Mathematica* programmer might solve a problem. We will not attempt to write the most efficient or shortest code here, but will instead concentrate on the process of programming – things like rewording the problem, finding the right tools and approach, and testing your program. Hopefully, this prelude will give you a better sense of the breadth and depth of the *Mathematica* language and help you answer the question, "Why use *Mathematica* to write programs?"

The second aim of this chapter is to help you become familiar with the *Mathematica* environment, showing you how to get started, how to work with the interface, documentation, and programming tools, and how to start putting them together to do interesting things. The basic syntax used by *Mathematica* is introduced, including functions and lists and several alternate syntaxes that you can use to input expressions. This is followed by information on how to get help when you get stuck or have trouble understanding an error that has occurred. If you are already familiar with these aspects of *Mathematica*, feel free to skim or skip these topics and return to them when the need arises.

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-1-107-11666-5 - Essentials of Programming in Mathematica®
Paul Wellin
Excerpt
More information

1.1 Introduction to programming

Computer programs are detailed and explicit descriptions of the steps to take to accomplish a specific task. The key is converting the original statement of the problem from something you might describe to a colleague in your native natural language into something that a computer can understand and operate on. As an example, suppose you want to write a program for your home's programmable thermostat to control the temperature in your house. A description such as "turn on the heat when it is cold and turn on the air conditioner when it is hot" may be entirely understandable to the humans in your household, but will be of little help in communicating with the thermostat. Instead, imagine formulating the problem as follows: if the ambient temperature drops below 17°C, turn on the heater until the ambient temperature reaches 22°C; if the temperature rises above 29°C, turn on the air conditioner until the temperature drops below 24°C. With this formulation you have enough to translate those instructions into a program.

Let's take this thought experiment one step further. What if you are away at work or school and a door in your house blew open causing the heater to stay on in an attempt to warm the house while cold air rushes in. Although this scenario is atypical, your program could include a conditional check that turns the unit off for one hour say, if it has been on continuously for over two hours.

The task that we commonly think of as "programming" is the set of steps that take a description like the explicit formulation above for the thermostat and actually write code in a language that can be executed on a processor on your computer or on the thermostat itself (with an embedded application). We will pick it up from here with an example that walks through the steps of creating, testing, and refining a program. Using a problem that is simple to describe and does not require a lot of sophisticated programming will help to better focus on these "meta" steps.

Your first Mathematica program

We will create a small program to solve a specific problem – finding and counting palindromic numbers. The process involves stating and then reformulating the problem, implementation, checking typical and atypical input, analyzing and improving efficiency, and performing a post-mortem. There are more aspects that will be discussed throughout this book, such as localization, options, argument checking, and documentation, but at this stage we will keep this initial problem simple. You are not expected to be familiar with all aspects of the program at this point; instead, try to focus on the process here, rather than the details.

As you read through the problem, enter each of the inputs in a *Mathematica* notebook exactly as they appear here, being careful about the syntax. When you have completed each input, press SHIFT+ENTER on your keyboard to evaluate, then go on to the next input (see Section 1.2 for details).

Problem Find all palindromic numbers less than one thousand, then determine how many palindromes there are that are less than one million.

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-1-107-11666-5 - Essentials of Programming in Mathematica®
Paul Wellin
Excerpt
More information

Reformulate the problem A number is a palindrome if it is equal to the number formed by reversing its digits. For example, 1552551 is a palindrome, 1552115 is not. The problem is to create a function that checks if a number is palindromic (returns True if it is, False otherwise); then, from a list of the numbers one through *n*, find all those that pass this palindrome test.

We have restated the problem with an eye toward creating explicit instructions. The ability to do this comes from a knowledge of the constructs and paradigms present in our language, something that is learned over time and experience with that language.

Implementation Given an integer, first we will get a list of its digits, then reverse that list, and finally check that the reversed list is identical to the original list. Start by getting the digits of a number.

```
In[1]:= IntegerDigits[1 552 551]
Out[1]= {1, 5, 5, 2, 5, 5, 1}
```

Function names like `IntegerDigits` are spelled out in full, capitalizing the first letter of each complete word. Arguments to functions, 1552551 in this example, are enclosed in square brackets.

Next, reverse the digits.

```
In[2]:= Reverse[IntegerDigits[1 552 551]]
Out[2]= {1, 5, 5, 2, 5, 5, 1}
```

Check if the list of digits is equal to the list of reversed digits.

```
In[3]:= IntegerDigits[1 552 551] == Reverse[IntegerDigits[1 552 551]]
Out[3]= True
```

Turn this into a program/function that can be run/evaluated for any input.

```
In[4]:= PalindromeQ[n_] := IntegerDigits[n] == Reverse[IntegerDigits[n]]
```

Check a few numbers:

```
In[5]:= PalindromeQ[12 345 678]
Out[5]= False

In[6]:= PalindromeQ[9 991 999]
Out[6]= True
```

Check atypical input Try the function with input for which it was not meant, for example, a symbol.

```
In[7]:= PalindromeQ[f]
Out[7]= True
```

That is not handling bad input correctly. We should try to restrict the function to integer input (Section 6.2 discusses the framework for issuing warning messages when bad input is given).

```
In[8]:= Clear[PalindromeQ];
        PalindromeQ[n_Integer] := IntegerDigits[n] == Reverse[IntegerDigits[n]]
```

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-1-107-11666-5 - Essentials of Programming in Mathematica®
Paul Wellin
Excerpt
More information

Check a few bad inputs:

```
In[10]:= PalindromeQ[f]
Out[10]= PalindromeQ[f]

In[11]:= PalindromeQ[{a, b, c}]
Out[11]= PalindromeQ[{a, b, c}]
```

Solve original problem The original questions were: find all palindromes below one thousand and determine how many there are below one million.

```
In[12]:= Select[Range[10^3], PalindromeQ]
Out[12]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 22, 33, 44, 55, 66, 77, 88, 99, 101, 111, 121, 131, 141,
  151, 161, 171, 181, 191, 202, 212, 222, 232, 242, 252, 262, 272, 282, 292, 303, 313,
  323, 333, 343, 353, 363, 373, 383, 393, 404, 414, 424, 434, 444, 454, 464, 474, 484,
  494, 505, 515, 525, 535, 545, 555, 565, 575, 585, 595, 606, 616, 626, 636, 646, 656,
  666, 676, 686, 696, 707, 717, 727, 737, 747, 757, 767, 777, 787, 797, 808, 818, 828,
  838, 848, 858, 868, 878, 888, 898, 909, 919, 929, 939, 949, 959, 969, 979, 989, 999}

In[13]:= Count[Range[10^6], p_ /; PalindromeQ[p]]
Out[13]= 1998
```

Efficiency How fast is this function for a large list of integers?

```
In[14]:= AbsoluteTiming[
          Select[Range[10^6], PalindromeQ];
         ]
Out[14]= {2.50386, Null}
```

Not too bad, but perhaps we can speed things up. In the definition of `PalindromeQ`, we compute `IntegerDigits[n]` twice, once on each side of the equality check. Instead we can create a local variable `digs` and initialize it with the value of `IntegerDigits[n]`, thus only doing that computation once.

```
In[15]:= Clear[PalindromeQ];

In[16]:= PalindromeQ[n_Integer] := With[{digs = IntegerDigits[n]},
           digs == Reverse[digs]
          ]

In[17]:= AbsoluteTiming[
          Select[Range[10^6], PalindromeQ];
         ]
Out[17]= {2.31562, Null}
```

That didn't help much – probably introducing the scoping construct `With` negated any small gains from reducing the size of the computation. Another strategy might consider using strings

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-1-107-11666-5 - Essentials of Programming in Mathematica®
Paul Wellin
Excerpt
More information

which, we know from experience, can be very fast to work with. We will save this approach for Chapter 7 where we discuss strings and compare implementations using lists versus strings.

Actually this problem will parallelize well as the list of numbers to be checked can be distributed across kernels and processors. Any speed improvement will be tied to the number of processors available to run the kernels.

```
In[18]:= LaunchKernels[]
Out[18]= {KernelObject[1, local], KernelObject[2, local],
          KernelObject[3, local], KernelObject[4, local]}

In[19]:= DistributeDefinitions[PalindromeQ]
Out[19]= {PalindromeQ}

In[20]:= AbsoluteTiming[
          Parallelize[Select[Range[10^6], PalindromeQ]];
         ]
Out[20]= {0.887734, Null}
```

Postscript As you work through this example note that the notebook in which the code was developed and tested becomes a documentation of sorts with comments in text cells interspersed here and there to help whoever looks at the code to understand your thinking (including yourself several months or years later). In other words, the notebook interface is your development environment. You could also bundle up your code into a formal package – a platform-independent text file containing *Mathematica* commands – that could be loaded when needed (see Chapter 10).

The code in these examples uses a variety of programming styles and constructs that may be new to you. It is entirely forgivable if you do not understand them at this point – that is what this book is all about! By the time you have worked through a good deal of the book you should be comfortable solving problems using the basic principles of programming – making assignments, defining rules, using conditionals, recursion, and iteration.

Programming paradigms

As you start to write programs in *Mathematica*, it is natural to ask, "How does it compare with programming in other languages?" To start, there are certain details shared by all programming languages and environments. In any language, there is a finite collection of objects that must be put together in a specific order to make statements that are valid in that language. In natural languages, say English, those objects are the alphabet, punctuation symbols, spaces, and so on. The rules of the language describe what are valid statements in that language. In computer languages, those statements are used to communicate instructions to a computer, say to add two integers, or to print a string to the screen, or to iterate a function until a condition is met.

The chief way in which languages differ is in the style of programming that each language uses. Early programming languages, such as FORTRAN and C (and a bit later PERL, PYTHON, and many

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-1-107-11666-5 - Essentials of Programming in Mathematica®
Paul Wellin
Excerpt
More information

others), use what is referred to as an *imperative* style of programming. This is one in which a specific sequence of operations are explicitly given and the flow of execution is controlled by the programmer. In other words, the instructions describe *how to perform* each step.

Declarative languages, in contrast, describe the desired result instead of focusing on the underlying machine instructions to get there. That is, they describe *what to do*, rather than how. LISP, SCHEME, HASKELL, and *Mathematica* are mostly declarative languages, although each allows an imperative-style of programming (particularly procedural) to be used as well.

Further distinguishing languages from one another are the steps the programmer takes to try out their programs. In most traditional languages such as FORTRAN, C, and JAVA, you start by writing code in an editor using the constructs of the language. In JAVA, you might write the following source code in an editor:

```
public class HelloWorld {
  public static void main (String[] args) {
    System.out.println ("Hello, World");
   }
 }
```

For a similar program in C, you would type:

```
main () {
  printf ("hello, world");
 }
```

After saving your code in an appropriately named file, you then run it through a compiler to create an executable file that can be run on your computer. If there is an error somewhere in your program, you go back to your source code and, perhaps with the aid of a debugger, you find and correct any errors and then recompile your program and execute it again.

But there is another way. Some languages, such as LISP, PERL, PYTHON, RUBY, and *Mathematica*, are referred to as *interpreted languages*. What this means is that you use them in an interactive mode: you type in a command, evaluate it in place, and the result is returned. The environment in which you type in the command and see the results is the same; in *Mathematica*, that environment is the notebook.

In PERL, the "Hello World" program is simply:

```
print "Hello World!\n";
```

In *Mathematica*, you type in and then evaluate the following:

```
In[21]:= Print["Hello world"]
Hello world
```

The advantage of interpreted languages is that you entirely avoid the compile/run/debug cycle of other languages. In addition, these so-called *high-level* interpreted languages contain functions for performing many tasks that you would typically have to implement in a lower-level language (if you are a C programmer, compare `Reverse[lis]` in *Mathematica* with a C program to do the same

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-1-107-11666-5 - Essentials of Programming in Mathematica®
Paul Wellin
Excerpt
More information

thing). This saves you time and lets you focus more on the problem at hand and less on the intricacies of memory management, libraries, and registers say.

In the old days, the advantages of compiled languages were speed and access to low-level aspects of your computer. But with advances in hardware, these seeming advantages are much less clear. If a program runs a few tenths of a second faster but takes many more minutes to create, your productivity gains start to evaporate pretty quickly.

Creating programs

In any language, the creation of computer programs involves many different tasks. For our purposes here it is useful to distinguish two broad categories: programming and software development.

Programming involves:

- *Analysis of the problem*. This includes understanding precisely what is being asked and what the answers will look like.
- *Restatement of the problem*. Phrasing the problem in a way that is closer to how you will actually write the program; creating pseudo-code.
- *Formulating a plan of attack*. With what type of data can you prototype the problem? Is the problem one that can be solved using list manipulation? Should you use string functions? Will the computation require extraordinary resources?
- *Implementation*. Translating the problem into code; modularization; if using an interactive language, trying out pieces as you go.

Software development includes:

- *Verification of correctness*. Checking solutions for many types of input; comparing with known solutions; using different algorithms to make comparisons.
- *Debugging*. Finding and correcting errors – from bad syntax or from incorrect algorithms.
- *Robustness*. Checking atypical input that includes special cases, bad input, and generally input that your program was not designed for.
- *Performance and efficiency*. Identifying bottlenecks (possibly through profiling) to improve memory usage and reduce compute time, network bandwidth.
- *Documentation*. Comments and notes about why a specific function or approach was used at a certain step; documenting lightly as you program and then adding more substantial information later.
- *User interface*. What does the user need to know to understand how to use your program? Does it behave like built-in functions or will the user need to learn a new interface element.
- *Portability*. Does your program work on different platforms, environments? Do you need to compile for different operating systems? (Your *Mathematica* programs will run unchanged on any platform that runs *Mathematica*.)

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-1-107-11666-5 - Essentials of Programming in Mathematica®
Paul Wellin
Excerpt
More information

- *Code maintenance*. Periodic checks that your code runs in the latest version of your programming language, on the latest operating system, etc.

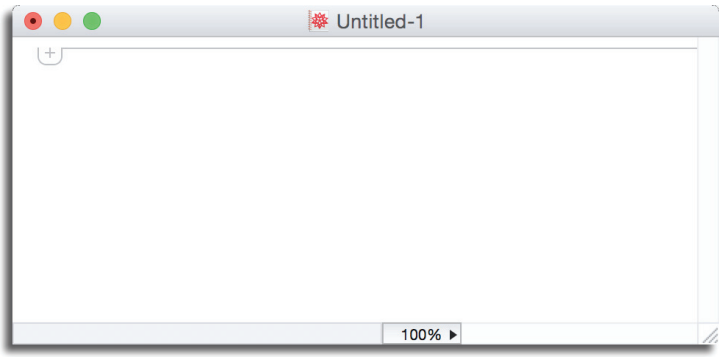
Not every program you create will include all of the above pieces, and the order in which you implement them is not so well-defined and discrete as stated here. For example, the `PalindromeQ` program created in the previous section did not include formal documentation outside of comments in the notebook itself. Smaller programs written to solve a basic problem, or programs that you have no intention of sharing with a colleague, student, or client, may not need some of the niceties (such as user interface elements) that another program would have. But the availability of these programming tools and constructs makes *Mathematica* a good choice for both small programs and large-scale applications that will be used by others. This book is designed to give you a good sense of this breadth and depth and to provide a foundational set of tools to use in your *Mathematica* programming.

1.2 Getting started

Let us now turn to some of the basics needed to start using *Mathematica*. If you are not familiar with *Mathematica*, it would be helpful to try out the examples in this section before going further. These examples should give you a sense of what it means to enter, evaluate, and work with some simple computations. If you are already familiar with *Mathematica*, feel free to skim this section lightly.

Starting and running Mathematica

After launching *Mathematica*, parts of it will load into memory and soon a blank window will appear on the screen. This window, called a notebook, is the visual interface to *Mathematica*.



When a blank notebook first appears on the screen, either from just starting *Mathematica* or from selecting New in the File menu, you can start typing immediately. For example, type `N[Pi, 200]` and then press SHIFT+ENTER (hold down the Shift key while pressing the Enter key) to evaluate the expression. *Mathematica* will evaluate the result and print the 200-decimal-digit approximation to π.

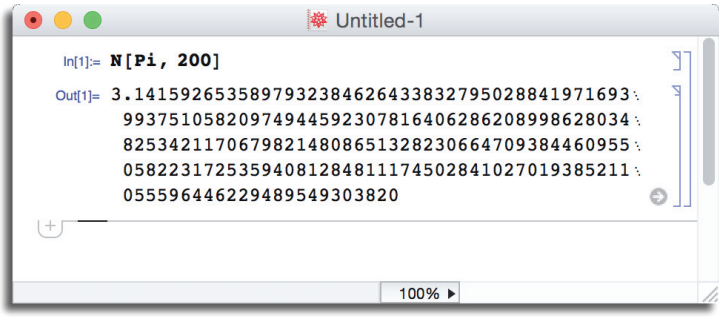
When you evaluate an expression in a notebook, *Mathematica* automatically adds input and output prompts *after* you evaluate your input. In the example notebook at the top of the next page,

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-1-107-11666-5 - Essentials of Programming in Mathematica®
Paul Wellin
Excerpt
More information

these are denoted In[1]:= and Out[1]=. These prompts can be thought of as markers (or labels) that you can refer to during your *Mathematica* session.



New input can be entered whenever there is a horizontal line that runs across the width of the notebook. If one is not present where you wish to place an input cell, move the cursor up and down until it changes to a horizontal bar and then click the mouse once. A horizontal line should appear across the width of the window. You can immediately start typing and an input cell will be created.

Mathematical expressions can be entered in a traditional-looking two-dimensional format using either palettes for quick entry of template expressions, or keyboard equivalents. For example, the following expression can be entered by using the Basic Math Assistant palette (under the Palettes menu), or through a series of keystrokes as described in the tutorial Entering Two-Dimensional Input in the Wolfram Language Documentation Center (WLDC).

```
In[1]:= 2^100
Out[1]= 1 267 650 600 228 229 401 496 703 205 376
```

To refer to the result of the previous calculation, use the symbol %.

```
In[2]:= % + 1
Out[2]= 1 267 650 600 228 229 401 496 703 205 377
```

To refer to the result of any earlier calculation, use its Out[*i*] label or, equivalently, % *i*.

```
In[3]:= Out[1]
Out[3]= 1 267 650 600 228 229 401 496 703 205 376

In[4]:= (%1) / 2^90
Out[4]= 1024
```

Mathematical expressions

Mathematical expressions can be entered in a linear syntax using arithmetic operators common to almost all computer languages.

```
In[5]:= 39 / 13
Out[5]= 3
```

Enter this expression in the traditional form by typing 39, CTRL /, then 13.

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-1-107-11666-5 - Essentials of Programming in Mathematica®
Paul Wellin
Excerpt
More information


```
In[6]:= 39/13
Out[6]= 3
```

The caret (^) is used for exponentiation.

```
In[7]:= 2^5
Out[7]= 32
```

To enter this expression in a more traditional typeset form, type 2, CTRL[^], and then 5.

```
In[8]:= 2^5
Out[8]= 32
```

Multiplication can be indicated by putting a space between the two factors, as in mathematical notation. *Mathematica* will automatically display the traditional multiplication sign, ×, between two numbers. The asterisk (*) is also used for that purpose, as is traditional in most computer languages.

```
In[9]:= 2 × 5
Out[9]= 10

In[10]:= 2 * 5
Out[10]= 10
```

Operations are given the same precedence as in mathematics. In particular, multiplication and division have a higher precedence than addition and subtraction: $3 + 4 \times 5$ equals 23 and not 35.

```
In[11]:= 3 + 4 × 5
Out[11]= 23
```

You can enter typeset expressions in several different ways: directly from the keyboard as we did above, using a long (functional) form, or via palettes available from the Palettes menu. Table 1.1 shows some of the more commonly used typeset expressions and how they are entered through the keyboard. Try to become comfortable entering these inputs so that you can easily enter the kinds of expressions used in this book.

TABLE 1.1. *Entering typeset expressions*

Display form	Long (functional) form	Keystrokes
x^2	`Superscript[x,2]`	x, CTRL+6, 2
x_i	`Subscript[x,i]`	x, CTRL+_, i
$\frac{x}{y}$	`FractionBox[x,2]`	x, CTRL+/, y
$\sqrt{x}$	`SqrtBox[x]`	CTRL+2, x
$x \geq y$	`GreaterEqual[x,2]`	x, ESC, >=, ESC, y

© in this web service Cambridge University Press www.cambridge.org