4 Functional programming

Programming in *Mathematica* is essentially a matter of writing user-defined functions that work like mathematical functions; when applied to specific values, they perform computations producing results. In fact, these functions can operate on arbitrary expressions, including other functions. This *functional* style of programming distinguishes *Mathematica* from more traditional procedural languages like C and Fortran, and a facility at functional programming is essential for taking full advantage of *Mathematica*'s powerful language to solve your computational tasks.

4.1 Introduction

Functions are objects that operate on expressions and output unique expressions for each input. We can think of functions as mathematicians do. For example, here is a definition for a function of two variables.

```
In[1]:= f[x_, y_] := Cos[x] + Sin[y]
```

You can evaluate the function for numeric or symbolic values.

```
In[2]:= f[π, 1.6]
Out[2]= -0.000426397

In[3]:= f[θ, ρ]
Out[3]= Cos[θ] + Sin[ρ]
```

Functions can be significantly more complicated objects. Below is a function that operates on functions. Like the function `f` above it takes two arguments, but, in this case, its arguments are a function or expression, and a list containing the variable of integration and the integration limits.

```
In[4]:= Integrate[Exp[I π x], {x, a, b}]
Out[4]= i (e^(i a π) - e^(i b π)) / π
```

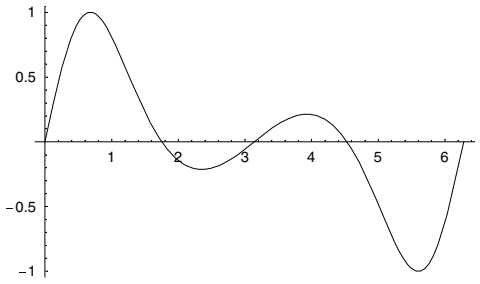
This particular function can be also be called with a function and a variable.

```
In[5]:= Integrate[Exp[I π x], x]

Out[5]= -(i e^(i π x) / π)
```

Here is a function that also takes two arguments and operates on functions, but it returns a graphical object as its value.

```
In[6]:= Plot[Sin[x + √2 Sin[x]], {x, 0, 2 π}]
```




```
Out[6]= - Graphics -
```

Programming involves writing a set of instructions to be applied for some appropriate input. Whereas procedural programs provide a step-by-step set of instructions, functional programming involves the application of functions to their arguments. For example, here is a traditional procedural approach to switching the elements in a list of pairs.

```
In[7]:= lis = {{α, 1}, {β, 2}, {γ, 3}}

Out[7]= {{α, 1}, {β, 2}, {γ, 3}}

In[8]:= temp = lis;
        Do[{temp[[i, 1]], temp[[i, 2]]} = {lis[[i, 2]], lis[[i, 1]]},
          {i, 1, Length[lis]}];
        temp

Out[10]= {{1, α}, {2, β}, {3, γ}}
```

We first allocate an empty array `temp`, of the same size as `lis`; then we put elements into `temp` one by one as we loop over `lis`; finally we return the value of `temp`.

Here is a simpler procedure using a structured iteration.

```
In[11]:= Table[{lis[[i, 2]], lis[[i, 1]]}, {i, 1, 3}]

Out[11]= {{1, α}, {2, β}, {3, γ}}
```

And here is a functional approach to solving the same problem.

```
In[12]:= Map[Reverse, lis]
Out[12]= {{1, α}, {2, β}, {3, γ}}
```

This simple example illustrates several of the key features of functional programming. A functional approach often allows for a more direct implementation of the solution to many problems, especially when list manipulations are involved. Notice that the procedural approach required setting up a list structure and then looping over the list as `i` takes on successive values, whereas the functional approach simply applied the `Reverse` function to the list directly.

Up to this point, we have described fairly simple functions and stayed focused on the built-in functions present in *Mathematica*. In this chapter we will first take a look at some of the most powerful and useful functional programming constructs in *Mathematica* and then discuss the creation of our own functions, using many of the list and string manipulating functions discussed earlier. It is well worthwhile to spend time familiarizing yourself with these functions by playing around with them; for example, create various lists and apply built-in functions to them. Having a larger vocabulary of built-in functions will not only make it easier to follow the programs and do the exercises here, but will enhance your own programming skills as well.

4.2 Functions for manipulating expressions

Three of the most powerful and commonly used functions by experienced *Mathematica* programmers are `Map`, `Apply`, and `Thread`. They provide very sophisticated ways of manipulating expressions in *Mathematica*. Becoming familiar with them is essential to functional programming in *Mathematica*. In this section we will discuss their syntax and look at some simple examples of their use. We will also briefly look at some related functions (`Inner` and `Outer`), which will prove useful in manipulating the structure of your expressions. These higher-order functions will be used throughout the rest of this book.

Map

`Map` applies a function to each element in a list.

```
In[1]:= Map[Head, {3, 22/7, π}]

Out[1]= {Integer, Rational, Symbol}
```

This is illustrated using an undefined function *f* and a simple linear list.

```
In[2]:= Map[f, {a, b, c}]

Out[2]= {f[a], f[b], f[c]}
```

More generally, mapping a function `f` over the expression `g[a,b,c]` essentially wraps the function `f` around each of the *elements* of `g`.

```
In[3]:= Map[f, g[a, b, c]]

Out[3]= g[f[a], f[b], f[c]]
```

So this general computation is identical to `Map[f,{a,b,c}]`, except in that example `g` is replaced with `List` (remember that `FullForm[{a,b,c}]` is `List[a,b,c]`).

The real power of the `Map` function is that you can map *any* function across any expression for which that function makes sense. Using the `Reverse` function with `Map`, you can reverse the order of elements in each list of a nested list.

```
In[4]:= Map[Reverse, {{a, b}, {c, d}, {e, f}}]

Out[4]= {{b, a}, {d, c}, {f, e}}
```

The elements in each of the inner lists in a nested list can be sorted.

```
In[5]:= Map[Sort, {{2, 6, 3, 5}, {7, 4, 1, 3}}]

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

Often, you will need to define your own function to perform some computation on every element of a list. This is the sort of computation that `Map` is expressly designed for. Here is a list of three elements.

```
In[6]:= vec = {2, π, γ};
```

If we wished to square each element and add 1, we could first define a function that performs this computation on its arguments.

```
In[7]:= f[x_] := x^2 + 1
```

Mapping this function over `vec`, will then wrap `f` around each element and evaluate `f` of those elements.

```
In[8]:= Map[f, vec]
Out[8]= {5, 1 + π^2, 1 + γ^2}
```

Later in this chapter we will look at even simpler ways of performing such computations.

Thread and MapThread

The `Thread` function exchanges operations with arguments that are lists.

```
In[9]:= Thread[g[{a, b, c}, {x, y, z}]]
Out[9]= {g[a, x], g[b, y], g[c, z]}
```

You can accomplish something quite similar with `MapThread`. It differs from `Thread` in that it takes two arguments – the function that you are mapping and a list of two (or more) lists as arguments of the function. It creates a new list in which the corresponding elements of the old lists are paired (or zipped together).

```
In[10]:= MapThread[g, {{a, b, c}, {x, y, z}}]
Out[10]= {g[a, x], g[b, y], g[c, z]}
```

With `Thread`, you can fundamentally change the structure of the expressions you are working with. For example, this threads the `Equal` function over the two lists given as its arguments.

```
In[11]:= Thread[Equal[{a, b, c}, {x, y, z}]]
Out[11]= {a == x, b == y, c == z}

In[12]:= Map[FullForm, %]
Out[12]= {Equal[a, x], Equal[b, y], Equal[c, z]}
```

Here is another example of the use of `Thread`. We start off with a list of variables and a list of values.

```
In[13]:= vars = {x1, x2, x3, x4, x5};

In[14]:= values = {1.2, 2.5, 5.7, 8.21, 6.66};
```

From these two lists, we create a list of rules.

```
In[15]:= Thread[Rule[vars, values]]
Out[15]= {x1 → 1.2, x2 → 2.5, x3 → 5.7, x4 → 8.21, x5 → 6.66}
```

Notice how we started with a *rule of lists* and `Thread` produced a *list of rules*. In this way, you might think of `Thread` as a generalization of `Transpose`.

Here are a few more examples of `MapThread`. This raises each element in the first list to the power given by the corresponding element in the second list.

```
In[16]:= MapThread[Power, {{2, 6, 3}, {5, 1, 2}}]

Out[16]= {32, 6, 9}
```

Using `Trace`, you can view some of the intermediate steps that *Mathematica* performs in doing this calculation.

```
In[17]:= MapThread[Power, {{2, 6, 3}, {5, 1, 2}}] // Trace

Out[17]= {MapThread[Power, {{2, 6, 3}, {5, 1, 2}}],
          {2^5, 6^1, 3^2}, {2^5, 32}, {6^1, 6}, {3^2, 9}, {32, 6, 9}}
```

Using the `List` function, the corresponding elements in the three lists are placed in a list structure (note that `Transpose` would do the same thing).

```
In[18]:= MapThread[List, {{5, 3, 2}, {6, 4, 9}, {4, 1, 4}}]

Out[18]= {{5, 6, 4}, {3, 4, 1}, {2, 9, 4}}
```

The Listable attribute

Many of the built-in functions that take a single argument have the property that, when a list is the argument, the function is automatically applied to all of the elements in the list. In other words, these functions are automatically mapped on to the elements of the list. For example, the `Log` function has this attribute.

```
In[19]:= Log[{a, E, 1}]

Out[19]= {Log[a], 1, 0}
```

This is the same result you get using the `Map` function.

```
In[20]:= Map[Log, {a, E, 1}]

Out[20]= {Log[a], 1, 0}
```

Many of the built-in functions that take two or more arguments have the property that, when multiple lists are the arguments, the function is automatically applied to all of the corresponding elements in the list. In other words, these functions are automatically threaded on to the elements of the list.

```
In[21]:= {4, 6, 3} + {5, 1, 2}

Out[21]= {9, 7, 5}
```

This gives the same result as using the `Plus` function with `MapThread`.

```
In[22]:= MapThread[Plus, {{4, 6, 3}, {5, 1, 2}}]

Out[22]= {9, 7, 5}
```

Functions that are either automatically mapped or threaded on to the elements of list arguments are said to be `Listable`. Many of *Mathematica*'s built-in functions have this `Attribute`.

```
In[23]:= Attributes[Log]

Out[23]= {Listable, NumericFunction, Protected}

In[24]:= Attributes[Plus]

Out[24]= {Flat, Listable, NumericFunction,
           OneIdentity, Orderless, Protected}
```

By default, functions that you define do not have any attributes associated with them. So, for example, if you define a function `g`, say, it will not automatically be threaded over a list.

```
In[25]:= g[{{a, b}, {c, d}}]

Out[25]= g[{{a, b}, {c, d}}]
```

If you want your function to have the ability to thread over lists, give it the `Listable` attribute using `SetAttributes`.

```
In[26]:= SetAttributes[g, Listable]

In[27]:= g[{{a, b}, {c, d}}]

Out[27]= {{g[a], g[b]}, {g[c], g[d]}}
```

Note that clearing a symbol only clears values associated with that symbol. It does not clear any attributes associated with the symbol.

```
In[28]:= Clear[g]

In[29]:= ?g

Global`g

 Attributes[g] = {Listable}
```

To clear attributes, you need to use `Remove`.

```
In[30]:= Remove[g]
```

Now there is no remaining information associated with g.

```
In[31]:= ? g

         Information::notfound : Symbol g not found. More…
```

Apply

Whereas Map is used to perform the same operation on each element of an expression, Apply is used to change the structure of an expression.

```
In[32]:= Apply[f, g[a, b, c]]

Out[32]= f[a, b, c]
```

The function f was applied to the expression g[a,b,c] and Apply replaced the head of g[a,b,c] with f.
If the second argument is a list, applying f to that expression simply replaces its head (List) with f.

```
In[33]:= Apply[f, {a, b, c}]

Out[33]= f[a, b, c]
```

The following computation shows the same thing, except we are using the internal represen tation of the list {a,b,c} here to better see how the structure is changed.

```
In[34]:= Apply[f, List[a, b, c]]

Out[34]= f[a, b, c]
```

We see that the elements of List are now the arguments of f. Essentially, you should think of Apply[*f* , *expr*] as replacing the head of *expr* with *f* .

```
In[35]:= Apply[Plus, {1, 2, 3, 4}]

Out[35]= 10
```

Here, List[1,2,3,4] has been changed to Plus[1,2,3,4] or, in other words, the head List has been replaced by Plus.

Plus[a,b,c,d] is the internal representation of the sum of these four symbols that you would normally write a+b+c+d.

```
In[36]:= Plus[a, b, c, d]

Out[36]= a + b + c + d
```

This list conversion can be applied to an entire list.

```
In[37]:= Apply[f, {{1, 2, 3}, {5, 6, 7}}]
Out[37]= f[{1, 2, 3}, {5, 6, 7}]
```

This is just vector addition.

```
In[38]:= Apply[Plus, {{1, 2, 3}, {5, 6, 7}}]
Out[38]= {6, 8, 10}
```

One important distinction between `Map` and `Apply` that you should be aware of concerns the level of the expression at which each operate. By default, `Map` operates at level 1. That is, in `Map[`*f*, *expr*`]`, *f* will be applied to each element at the top level of *expr*. So, for example, if *expr* consists of a nested list, *f* will be applied to each of the sublists, but not deeper, by default.

```
In[39]:= Map[f, {{a, b}, {c, d}}]
Out[39]= {{1 + a^2, 1 + b^2}, {1 + c^2, 1 + d^2}}
```

If you wish to apply `f` at a deeper level, then you have to specify that explicitly using a third argument to `Map`.

```
In[40]:= Map[f, {{a, b}, {c, d}}, {2}]
Out[40]= {{1 + a^2, 1 + b^2}, {1 + c^2, 1 + d^2}}
```

`Apply`, on the other hand, operates at level 0. That is, in `Apply[`*f*, *expr*`]`, `Apply` looks at the part 0 of *expr* (that is, its `Head`) and replaces it with *f*.

```
In[41]:= Apply[f, {{a, b}, {c, d}}]
Out[41]= f[{a, b}, {c, d}]
```

Again, if you wish to apply `f` at a different level, then you have to specify that explicitly using a third argument to `Apply`.

```
In[42]:= Apply[f, {{a, b}, {c, d}}, 1]
Out[42]= {f[a, b], f[c, d]}
```

For example, to apply `Plus` to each of the inner lists, you need to specify that `Apply` will operate at level 1.

```
In[43]:= Apply[Plus, {{1, 2, 3}, {5, 6, 7}}, {1}]
Out[43]= {6, 18}
```

If you are a little unsure of what has just happened, consider the following example and, instead of `f`, think of `Plus`.

```
In[44]:= Apply[f, {{1, 2, 3}, {5, 6, 7}}, {1}]
Out[44]= {f[1, 2, 3], f[5, 6, 7]}
```

Inner and Outer

The `Outer` function applies a function to all of the combinations of the elements in several lists. This is a generalization of the mathematical *outer product*.

```
In[45]:= Outer[f, {a, b}, {2, 3, 4}]
Out[45]= {{f[a, 2], f[a, 3], f[a, 4]}, {f[b, 2], f[b, 3], f[b, 4]}}
```

Using the `List` function as an argument, you can create lists of ordered pairs that combine the elements of several lists.

```
In[46]:= Outer[List, {a, b}, {2, 3, 4}]
Out[46]= {{{a, 2}, {a, 3}, {a, 4}}, {{b, 2}, {b, 3}, {b, 4}}}
```

Using `Inner`, you can thread a function on to several lists and then use the result as the argument to another function.

```
In[47]:= Inner[f, {a, b, c}, {d, e, f}, g]
Out[47]= g[f[a, d], f[b, e], f[c, f]]
```

This function lets you carry out some interesting operations.

```
In[48]:= Inner[Times, {x1, y1, z1}, {x2, y2, z2}, Plus]
Out[48]= x1 x2 + y1 y2 + z1 z2
```

```
In[49]:= Inner[List, {a, b, c}, {d, e, f}, Plus]
Out[49]= {a + b + c, d + e + f}
```

Looking at these two examples, you can see that `Inner` is really a generalization of the mathematical dot product.

```
In[50]:= Dot[{x1, y1, z1}, {x2, y2, z2}]
Out[50]= x1 x2 + y1 y2 + z1 z2
```

Exercises

1. Write a function `addPair[{x,y}]` that adds the elements in a pair. Then use your `addPair` function to sum each pair from the following.

```
data = {{1, 2}, {2, 3}, {3, 4}, {4, 5}, {5, 6}};
```

   Your output should look like `{3,5,7,9,11}`.

2. Use `Apply` to add the elements in each pair from a list of pairs of numbers such as in the previous exercise.

3. A matrix can be rotated by performing a number of successive operations. Rotate the matrix `{{1,2,3},{4,5,6}}` clockwise by 90 degrees, obtaining `{{4,1},{5,2},{6,3}}`, in two steps. Use `TableForm` to display the results.

4. While matrices can easily be added using `Plus`, matrix multiplication is more complicated. The `Dot` function, written as a single period, can be used.

```
In[1]:= {{1, 2}, {3, 4}}.{x, y}
Out[1]= {x + 2 y, 3 x + 4 y}
```

   Perform matrix multiplication on `{{1,2},{3,4}}` and `{x,y}` without using `Dot`. (This can be done in two or three steps.)

5. `FactorInteger[n]` returns a nested list of prime factors and their exponents for the number *n*.

```
In[2]:= FactorInteger[3628800]
Out[2]= {{2, 8}, {3, 4}, {5, 2}, {7, 1}}
```

   Use `Apply` to reconstruct the number from this nested list.

6. Repeat the above exercise but instead use `Inner` to construct the original number *n* from the factorization given by `FactorInteger[n]`.

7. Using `Inner`, write a function `div[vecs, vars]` that computes the divergence of an *n*-dimensional vector field *vecs* = $\{e_1, e_2, \ldots, e_n\}$ dependent upon *n* variables *vars* = $\{v_1, v_2, \ldots, v_n\}$. The divergence is given by the sum of the pairwise partial derivatives.

$$\frac{\partial e_1}{\partial v_1} + \frac{\partial e_2}{\partial v_2} + \ldots + \frac{\partial e_n}{\partial v_n}$$

4.3 Iterating functions

A commonly performed task in computer science and mathematics is to repeatedly apply a function to some expression. Iterating functions has a long and rich tradition in the history of computing. Perhaps the most famous example is Newton's method for root finding. Chaos theory rests on studying how iterated functions behave under small perturbations of their initial conditions or starting values. In this section, we will introduce several functions available in *Mathematica* for function iteration. In later chapters we will apply these and other programming constructs to look at some applications of iteration, including Newton's method.

The `Nest` function is used to iterate functions. Here, `g` is iterated (or applied to) `a` four times.

```
In[1]:= Nest[g, a, 4]

Out[1]= g[g[g[g[a]]]]
```

The `NestList` function displays all of the intermediate values of the `Nest` operation.

```
In[2]:= NestList[g, a, 4]

Out[2]= {a, g[a], g[g[a]], g[g[g[a]]], g[g[g[g[a]]]]}
```

Using a starting value of 0.85, this generates a list of ten iterates of the `Cos` function.

```
In[3]:= NestList[Cos, 0.85, 10]

Out[3]= {0.85, 0.659983, 0.790003, 0.703843, 0.76236, 0.723208,
         0.749687, 0.731902, 0.743904, 0.73583, 0.741274}
```

The list elements above are the values of `0.85`, `Cos[0.85]`, `Cos[Cos[0.85]]`, and so on.

```
In[4]:= {0.85, Cos[0.85], Cos[Cos[0.85]], Cos[Cos[Cos[0.85]]]}

Out[4]= {0.85, 0.659983, 0.790003, 0.703843}
```

In fact, the iterates of the cosine function tend towards a fixed point which can be obtained with `FixedPoint`. This function is particularly useful when you do not know how many iterations to perform on a function whose iterations eventually settle down.

```
In[5]:= FixedPoint[Cos, 0.85]

Out[5]= 0.739085
```

Whereas `Nest` and `NestList` operate on functions of one variable, `Fold` and `FoldList` generalize this notion by iterating a function of two arguments. In the following example, the function `f` is first applied to a starting value `x` and the first element from a

list, then this result is used as the first argument of the next iteration, with the second argument coming from the second element in the list, and so on.

```
In[6]:= Fold[f, x, {a, b, c}]
Out[6]= f[f[f[x, a], b], c]
```

If `FoldList` is used, then you will see all of the intermediate results of the `Fold` operation.

```
In[7]:= FoldList[f, x, {a, b, c}]
Out[7]= {x, f[x, a], f[f[x, a], b], f[f[f[x, a], b], c]}
```

It is easy to see what is going on with the `FoldList` function by working with an arithmetic operator. This generates "running sums."

```
In[8]:= FoldList[Plus, 0, {a, b, c, d}]
Out[8]= {0, a, a + b, a + b + c, a + b + c + d}

In[9]:= FoldList[Plus, 0, {1, 2, 3, 4, 5}]
Out[9]= {0, 1, 3, 6, 10, 15}
```

Exercises

1. Determine the locations after each step of a ten-step one-dimensional random walk. (Recall that you have already generated the step *directions* in Exercise 3 at the end of Section 3.2.)
2. Create a list of the step locations of a ten-step random walk on a square lattice.
3. Using `Fold`, create a function `fac[n]` that takes in an integer n as argument and returns the factorial of n; that is, $n(n-1)(n-2)\cdots 3\cdot 2\cdot 1$.

4.4 Programs as functions

A computer program is a set of instructions (a recipe) for carrying out a computation. When a program is evaluated with appropriate inputs, the computation is performed and the result is returned. In this sense, a program is a mathematical function and the inputs to a program are the arguments of the function. Executing a program is equivalent to applying a function to its arguments or, as it is often referred, making a function call.

User-defined functions

While there are a great many built-in functions in *Mathematica* that can be used to carry out computations, we invariably find ourselves needing customized functions. For example, once we have written a program to compute some values for some particular inputs, we might want to perform the same set of operations on different inputs. We would therefore like to create our own *user-defined* functions that we could then apply in the same way as we call a built-in function – by entering the function name and specific argument values. We will start with the proper syntax (or grammar) to use when writing a function definition.

The function definition looks very much like a mathematical equation: a left-hand side and a right-hand side separated by a colon-equal sign.

---

name`[`arg_1`_,`arg_2`_, ..., `arg_n`_]:=` *body*

The left-hand side starts with a symbol. This symbol is referred to as the *function name* (or sometimes just as the function, as in "the sine function"). The function name is followed by a set of square brackets, inside of which are a sequence of symbols ending with *blanks*. These symbols are referred to as the *function argument names*, or just the *function arguments*.

The right-hand side of a user-defined function definition is called the *body* of the function. The body can be either a single expression (a one-liner), or a series of expressions (a compound function), both of which will be discussed in detail shortly. Argument names from the left-hand side appear on the right-hand side without blanks. Basically, the right-hand side is a formula stating what computations are to be done when the function is called with specific values of the arguments.

When a user-defined function is defined with a delayed assignment (`:=`), nothing is returned. Thereafter, calling the function by entering the left-hand side of the function definition with specific values of the arguments causes the body of the function to be

computed with the specific argument values substituted where the argument names occur. In other words, when using delayed assignments, the body of your function is only evaluated when the function is called, not when it is first defined.

A simple example of a user-defined function is `square` which squares a value (it is a good idea to use a function name that indicates the purpose of the function).

```
In[1]:= square[x_] := x^2
```

After entering a function definition, you call the function in the same way that a built-in function is applied to an argument.

```
In[2]:= square[5]

Out[2]= 25
```

Building up programs

The ability to use the output of one function as the input of another is one of the keys to functional programming. A mathematician would call this "composition of functions." In *Mathematica*, this sequential application of several functions is known as a *nested function call*. Nested function calls are not limited to using a single function repeatedly, such as with the built-in `Nest` and `Fold` functions.

```
In[3]:= Cos[Sin[Tan[4.0]]]

Out[3]= 0.609053
```

To see the above computation more clearly, we can step through the computation.

```
In[4]:= Tan[4.0]

Out[4]= 1.15782

In[5]:= Sin[%]

Out[5]= 0.915931

In[6]:= Cos[%]

Out[6]= 0.609053
```

Wrapping the `Trace` function around the computation lets us see all of the intermediate expressions that are used in this evaluation.

```
In[7]:= Trace[Cos[Sin[Tan[4.0]]]]

Out[7]= {{{Tan[4.], 1.15782}, Sin[1.15782], 0.915931},
          Cos[0.915931], 0.609053}
```

You can read nested functions in much the same way that they are created, starting with the innermost functions and working towards the outermost functions. For example, the following expression determines whether all of the elements in a list are even numbers.

```
In[8]:= Apply[And, Map[EvenQ, {2, 4, 6, 7, 8}]]

Out[8]= False
```

Let us step through the computation much the same as *Mathematica* does, from the inside out.

1. Map the predicate `EvenQ` to every element in the list `{2,4,6,7,8}`.

```
In[9]:= Map[EvenQ, {2, 4, 6, 7, 8}]

Out[9]= {True, True, True, False, True}
```

2. Apply the logical function `And` to the result of the previous step.

```
In[10]:= Apply[And, %]

Out[10]= False
```

Finally, here is a definition that can be used on arbitrary lists.

```
In[11]:= setEvenQ[lis_] := Apply[And, Map[EvenQ, lis]]

In[12]:= setEvenQ[{11, 5, 1, 18, 16, 6, 17, 6}]

Out[12]= False
```

Another, more complicated, example returns the elements in a list of positive numbers that are bigger than all of the preceding numbers in the list.

```
In[13]:= Union[Rest[FoldList[Max, 0, {3, 1, 6, 5, 4, 8, 7}]]]

Out[13]= {3, 6, 8}
```

The `Trace` of the function call shows the intermediate steps of the computation.

```
In[14]:= Trace[Union[Rest[FoldList[Max, 0, {3, 1, 6, 5, 4, 8, 7}]]]]

Out[14]= {{{FoldList[Max, 0, {3, 1, 6, 5, 4, 8, 7}],
      {Max[0, 3], 3}, {Max[3, 1], Max[1, 3], 3},
      {Max[3, 6], 6}, {Max[6, 5], Max[5, 6], 6},
      {Max[6, 4], Max[4, 6], 6}, {Max[6, 8], 8},
      {Max[8, 7], Max[7, 8], 8}, {0, 3, 3, 6, 6, 6, 8, 8}},
     Rest[{0, 3, 3, 6, 6, 6, 8, 8}], {3, 3, 6, 6, 6, 8, 8}},
    Union[{3, 3, 6, 6, 6, 8, 8}], {3, 6, 8}}
```

This computation can be described as follows:

- The `FoldList` function is first applied to the function `Max`, `0`, and the list `{3,1,6,5,4,8,7}` (look at the `Trace` of this computation to see what `FoldList` is doing here).

```
In[15]:= FoldList[Max, 0, {3, 1, 6, 5, 4, 8, 7}]
Out[15]= {0, 3, 3, 6, 6, 6, 8, 8}
```

- The `Rest` function is then applied to the result of the previous step to remove the first element of the list.

```
In[16]:= Rest[%]
Out[16]= {3, 3, 6, 6, 6, 8, 8}
```

- Finally, the `Union` function is applied to the result of the previous step to remove duplicates.

```
In[17]:= Union[%]
Out[17]= {3, 6, 8}
```

Here is the function definition.

```
In[18]:= maxima[x_] := Union[Rest[FoldList[Max, 0, x]]]
```

Applying `maxima` to a list of numbers produces a list of all those numbers that are larger than any number that comes before it.

```
In[19]:= maxima[{4, 2, 7, 3, 4, 9, 14, 11, 17}]
Out[19]= {4, 7, 9, 14, 17}
```

Notice that in each of the nested functions described here, the argument of the first function was explicitly referred to, but the expressions that were manipulated in the succeeding function calls were not identified other than as the results of the previous steps (that is, as the results of the preceding function applications).

Here is an interesting application of building up a program with nested functions – the creation of a deck of cards. (*Hint*: The suit icons are entered by typing in `\[ClubSuit]`, `\[DiamondSuit]`, etc.)

```
In[20]:= cardDeck = Flatten[
           Outer[List, {♣, ♢, ♡, ♠}, Join[Range[2, 10], {J, Q, K, A}]], 1]

Out[20]= {{♣, 2}, {♣, 3}, {♣, 4}, {♣, 5}, {♣, 6}, {♣, 7}, {♣, 8}, {♣, 9}, {♣, 10},
          {♣, J}, {♣, Q}, {♣, K}, {♣, A}, {♢, 2}, {♢, 3}, {♢, 4}, {♢, 5}, {♢, 6},
          {♢, 7}, {♢, 8}, {♢, 9}, {♢, 10}, {♢, J}, {♢, Q}, {♢, K}, {♢, A}, {♡, 2},
          {♡, 3}, {♡, 4}, {♡, 5}, {♡, 6}, {♡, 7}, {♡, 8}, {♡, 9}, {♡, 10},
          {♡, J}, {♡, Q}, {♡, K}, {♡, A}, {♠, 2}, {♠, 3}, {♠, 4}, {♠, 5}, {♠, 6},
          {♠, 7}, {♠, 8}, {♠, 9}, {♠, 10}, {♠, J}, {♠, Q}, {♠, K}, {♠, A}}
```

You might think of `cardDeck` as a name for the expression given on the right-hand side of the immediate definition, or you might think of `cardDeck` as defining a function with zero arguments.

To understand what is going on here, we will build up this program from scratch. First we form a list of the number and face cards in a suit by combining a list of the numbers 2 through 10, `Range[2,10]`, with a four-element list representing the jack, queen, king, and ace, `{J,Q,K,A}`.

```
In[21]:= Join[Range[2, 10], {J, Q, K, A}]

Out[21]= {2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, A}
```

Now we pair each of the 13 elements in this list with each of the four elements in the list representing the card suits {♣,♢,♡,♠}. This produces a list of 52 ordered pairs representing the cards in a deck, where the king of clubs, for example, is represented by {♣,K}).

```
In[22]:= Outer[List, {♣, ♢, ♡, ♠}, %]

Out[22]= {{{♣, 2}, {♣, 3}, {♣, 4}, {♣, 5}, {♣, 6}, {♣, 7},
           {♣, 8}, {♣, 9}, {♣, 10}, {♣, J}, {♣, Q}, {♣, K}, {♣, A}},
          {{♢, 2}, {♢, 3}, {♢, 4}, {♢, 5}, {♢, 6}, {♢, 7}, {♢, 8},
           {♢, 9}, {♢, 10}, {♢, J}, {♢, Q}, {♢, K}, {♢, A}},
          {{♡, 2}, {♡, 3}, {♡, 4}, {♡, 5}, {♡, 6}, {♡, 7}, {♡, 8},
           {♡, 9}, {♡, 10}, {♡, J}, {♡, Q}, {♡, K}, {♡, A}},
          {{♠, 2}, {♠, 3}, {♠, 4}, {♠, 5}, {♠, 6}, {♠, 7}, {♠, 8},
           {♠, 9}, {♠, 10}, {♠, J}, {♠, Q}, {♠, K}, {♠, A}}}
```

While we now have all of the cards in the deck, they are grouped by suit in a nested list. We therefore un-nest the list:

```
In[23]:= Flatten[%, 1]

Out[23]= {{♣, 2}, {♣, 3}, {♣, 4}, {♣, 5}, {♣, 6}, {♣, 7}, {♣, 8}, {♣, 9}, {♣, 10},
          {♣, J}, {♣, Q}, {♣, K}, {♣, A}, {♢, 2}, {♢, 3}, {♢, 4}, {♢, 5}, {♢, 6},
          {♢, 7}, {♢, 8}, {♢, 9}, {♢, 10}, {♢, J}, {♢, Q}, {♢, K}, {♢, A}, {♡, 2},
          {♡, 3}, {♡, 4}, {♡, 5}, {♡, 6}, {♡, 7}, {♡, 8}, {♡, 9}, {♡, 10},
          {♡, J}, {♡, Q}, {♡, K}, {♡, A}, {♠, 2}, {♠, 3}, {♠, 4}, {♠, 5}, {♠, 6},
          {♠, 7}, {♠, 8}, {♠, 9}, {♠, 10}, {♠, J}, {♠, Q}, {♠, K}, {♠, A}}
```

Voila!

The step-by-step construction that we used here, applying one function at a time, checking each function call separately, is a very efficient way to *prototype* your programs in *Mathematica*. We will use this technique again in the next example.

We will perform what is called a *perfect shuffle*, consisting of cutting the deck in half and then interleaving the cards from the two halves. Rather than working with the large list of 52 ordered pairs during the prototyping, we will use a short made-up list. A short list of an even number of ordered integers is a good choice for the task.

```
In[24]:= d = Range[6]

Out[24]= {1, 2, 3, 4, 5, 6}
```

We first divide the list into two equal-sized lists.

```
In[25]:= Partition[d, Length[d] / 2]

Out[25]= {{1, 2, 3}, {4, 5, 6}}
```

We now want to interleave these two lists to form `{1,4,2,5,3,6}`. The first step is to pair the corresponding elements in each of the two lists above. This can be done using the `Transpose` function.

```
In[26]:= Transpose[%]

Out[26]= {{1, 4}, {2, 5}, {3, 6}}
```

We now un-nest the interior lists using the `Flatten` function. We could flatten our simple list using `Flatten[…]`, but, since we know that ultimately we will be dealing with ordered pairs rather than integers, we will use `Flatten[…,1]` as we did in creating the card deck.

```
In[27]:= Flatten[%, 1]

Out[27]= {1, 4, 2, 5, 3, 6}
```

That does the job. Given this prototype, it is easy to write the actual function to perform a perfect shuffle on a deck of cards. Notice we have generalized this shuffle to lists of arbitrary length.

```
In[28]:= shuffle[lis_] :=
          Flatten[Transpose[Partition[lis, Length[lis] / 2]], 1]

In[29]:= shuffle[cardDeck]

Out[29]= {{♣, 2}, {♡, 2}, {♣, 3}, {♡, 3}, {♣, 4}, {♡, 4}, {♣, 5}, {♡, 5}, {♣, 6},
          {♡, 6}, {♣, 7}, {♡, 7}, {♣, 8}, {♡, 8}, {♣, 9}, {♡, 9}, {♣, 10},
          {♡, 10}, {♣, J}, {♡, J}, {♣, Q}, {♡, Q}, {♣, K}, {♡, K}, {♣, A}, {♡, A},
          {♢, 2}, {♠, 2}, {♢, 3}, {♠, 3}, {♢, 4}, {♠, 4}, {♢, 5}, {♠, 5}, {♢, 6},
          {♠, 6}, {♢, 7}, {♠, 7}, {♢, 8}, {♠, 8}, {♢, 9}, {♠, 9}, {♢, 10},
          {♠, 10}, {♢, J}, {♠, J}, {♢, Q}, {♠, Q}, {♢, K}, {♠, K}, {♢, A}, {♠, A}}
```

Let us take this example one step further and construct a function that deals cards from a card deck. We will construct this function in stages using the prototyping method we showed earlier.
First we need to define a function that removes a single element from a randomly chosen position in a list.

```
In[30]:= removeRand[lis_] :=
          Delete[lis, Random[Integer, {1, Length[lis]}]]
```

The function `removeRand` first uses the `Random` function to randomly choose an integer *k* between 1 and the length of the list, and then uses the `Delete` function to remove the *k*th element of the list. For example, if a list has 10 elements, an integer between 1 and 10, say 6, is randomly determined and the element in the sixth position in the list is then removed from the list.

```
In[31]:= lis = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
         removeRand[lis]

Out[32]= {2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Now we want to make a function call that applies the `removeRand` function to the `cardDeck` list, then applies the `removeRand` function to the resulting list, then applies the `removeRand` function to the resulting list, and so on, a total of *n* times. The way to carry out this operation is with the `Nest` function.

```
Nest[removeRand, cardDeck, n]
```

Lastly, we want the cards that are removed from `cardDeck` rather than those that remain.

```
Complement[cardDeck, Nest[removeRand, cardDeck, n]]
```

Now, we write this up formally into the user-defined `deal` function.

```
In[33]:= deal[n_] := Complement[cardDeck, Nest[removeRand, cardDeck, n]]
```

Let us try it out.

```
In[34]:= deal[5]

Out[34]= {{♣, 3}, {♣, K}, {♢, 2}, {♡, K}, {♠, J}}
```

Not a bad hand!

Exercises

1. One of the games in the Illinois State Lottery is based on choosing *n* numbers, each between 0 and 9, with duplicates allowed; in practice, a selection is made from containers of numbered ping pong balls. We can model this game using a simple user-defined function, which we will call `pick` (after the official lottery names of *Pick 3* and *Pick 4*).

```
In[1]:= pick[n_] := Table[Random[Integer, {0, 9}], {n}]

In[2]:= pick[4]

Out[2]= {0, 9, 0, 4}
```

   This program can be generalized to perform *random sampling with replacement* on any list. Write a function `chooseWithReplacement[`*lis*`,`*n*`]`, where *lis* is the list, *n* is the number of elements being chosen and the following is a typical result.

```
In[3]:= chooseWithReplacement[{a, b, c, d, e, f, g, h}, 3]

Out[3]= {h, b, f}
```

2. Write your own user-defined functions using the `ToCharacterCode` and `FromCharacterCode` functions to perform the same operations as `StringInsert` and `StringDrop`.

3. Create a function `distance[`*a*`,`*b*`]` that finds the distance between two points *a* and *b* in the plane.

4. Write a user-defined function `interleave2` that interleaves the elements of two lists of unequal length. (You have already seen how to interleave lists of equal length using `Partition` earlier in this section.) Your function should take the lists `{1,2,3}` and `{a,b,c,d}` as inputs and return `{1,a,2,b,3,c,d}`.

5. Write a nested function call that creates a deck of cards and performs a perfect shuffle on it.

6. Write nested function calls using the `ToCharacterCode` and `FromCharacterCode` functions to perform the same operations as the built-in `StringJoin` and `StringReverse` functions.

4.5 Auxiliary functions

There are several major drawbacks to the `deal` function created in the previous section. In order to use `deal`, the definition of `removeRand` and the value of `cardDeck` must be entered before calling `deal`. It would be much more convenient if we could incorporate these functions within the `deal` function definition itself. In the next section, we will show how this can be done.

Compound functions

The left-hand side of a *compound function* is the same as that of any user-defined function. The right-hand side consists of consecutive expressions enclosed in parentheses and separated by semicolons.

---

name[*arg*$_1$_, *arg*$_2$_, …, *arg*$_n$_] := (*expr*$_1$; *expr*$_2$; …; *expr*$_m$)

The expressions can be user-defined functions (also known as *auxiliary* functions), value declarations, and function calls. When a compound function is evaluated with particular argument values, these expressions are evaluated in order and the result of the evaluation of the last expression is returned (by adding a semicolon after *expr*$_n$, the display of the final evaluation result can also be suppressed).

We will work with the `deal` function to illustrate how a compound function is created. We need the following three expressions.

```
In[1]:= cardDeck = Flatten[Outer[List,
            {♣, ◇, ♡, ♠}, Join[Range[2, 10], {J, Q, K, A}]], 1];

In[2]:= removeRand[lis_] :=
         Delete[lis, Random[Integer, {1, Length[lis]}]]

In[3]:= deal[n_] := Complement[cardDeck, Nest[removeRand, cardDeck, n]]
```

The conversion to a compound function is easily done. We will first remove the old definitions.

```
In[4]:= Clear[deal, cardDeck, removeRand]
```

Now we can create and enter the new definition.

```
In[5]:= deal[n_] := (
          cardDeck = Flatten[Outer[List,
             {♣, ♢, ♡, ♠}, Join[Range[2, 10], {J, Q, K, A}]], 1];
          removeRand[lis_] := Delete[lis,
            Random[Integer, {1, Length[lis]}]];
          Complement[cardDeck, Nest[removeRand, cardDeck, n]]
         )
```

Let us check that this works.

```
In[6]:= deal[5]

Out[6]= {{♣, 3}, {♢, 2}, {♡, 3}, {♡, 4}, {♡, Q}}
```

A couple of things should be pointed out about the right-hand side of a compound function definition. Since the expressions on the right-hand side are evaluated in order, value declarations and auxiliary function definitions should be given *before* they are used and the argument names used on the left-hand side of auxiliary function definitions *must* differ from the argument names used by the compound function itself.

Finally, when we enter a compound function definition, we are entering not only the function but also the auxiliary functions and the value declarations. If we then remove the function definition using `Clear`, the auxiliary function definitions and value declarations remain. This can cause a problem if we subsequently try to use the names of these auxiliary functions and values elsewhere.

So how does the global rule base treat compound functions? When a compound function definition is entered, a rewrite rule corresponding to the entire definition is created. Each time the compound function is subsequently called, rewrite rules are created from the auxiliary function definitions and value declarations within the compound function.

```
In[7]:= ?cardDeck

Global`cardDeck

cardDeck = {{♣, 2}, {♣, 3}, {♣, 4}, {♣, 5}, {♣, 6}, {♣, 7}, {♣, 8},
   {♣, 9}, {♣, 10}, {♣, J}, {♣, Q}, {♣, K}, {♣, A}, {♢, 2}, {♢, 3}, {♢, 4},
   {♢, 5}, {♢, 6}, {♢, 7}, {♢, 8}, {♢, 9}, {♢, 10}, {♢, J}, {♢, Q}, {♢, K},
   {♢, A}, {♡, 2}, {♡, 3}, {♡, 4}, {♡, 5}, {♡, 6}, {♡, 7}, {♡, 8}, {♡, 9},
   {♡, 10}, {♡, J}, {♡, Q}, {♡, K}, {♡, A}, {♠, 2}, {♠, 3}, {♠, 4}, {♠, 5},
   {♠, 6}, {♠, 7}, {♠, 8}, {♠, 9}, {♠, 10}, {♠, J}, {♠, Q}, {♠, K}, {♠, A}}
```

It is considered bad programming practice to leave auxiliary definitions in the global rule base that are not explicitly needed by the user of your function. In fact, it could interfere with a user's workspace and cause unintended problems.

To prevent these additional rewrite rules from being placed in the global rule base, you can localize their names by using the `Module` construct in the compound function definition. This is what we discuss next.

Localizing names: Module

When a user-defined function is written, it is generally a good idea to isolate the *names* of values and functions defined on the right-hand side from the outside world in order to avoid any conflict with the use of a name elsewhere in the session (for example, `cardDeck` might be used elsewhere to represent a pinochle deck). This can be done by wrapping the right-hand side of the function definition in the built-in `Module` function.

---

name`[`arg_1`_, `arg_2`_, ..., `arg_n`_] := Module[{`$name_1$`, `$name_2$` = `*value*`, ...}, `*expr*`]`

The first argument of the `Module` function is a list of the names we want to localize. If we wish, we can assign values to these names, as is shown with $name_2$ above (the assigned value is only an initial value and can be changed subsequently). The list is separated from the right-hand side by a comma and so the parentheses enclosing the right-hand side of a compound function are not needed.

We can demonstrate the use of `Module` with the `deal` function.

```
In[8]:= Clear[deal]

In[9]:= deal[n_] := Module[{cardDeck, removeRand},
          cardDeck = Flatten[Outer[List,
             {♣, ◇, ♡, ♠}, Join[Range[2, 10], {J, Q, K, A}]], 1];
          removeRand[lis_] := Delete[lis,
            Random[Integer, {1, Length[lis]}]];
          Complement[cardDeck, Nest[removeRand, cardDeck, n]]]
```

Briefly, when `Module` is encountered, the symbols that are being localized (`cardDeck` and `removeRand` in the above example) are temporarily given new and unique names and all occurrences of those symbols in the body of the `Module` are given those new names as well. In this way, these unique and temporary names, which are local to the function will not interfere with any functions outside of the `Module`.

It is generally a good idea to wrap the right-hand side of all compound function definitions in the `Module` function. Another way to avoid conflicts in the use of names of auxiliary function definitions is to use a function that can be applied without being given a name. Such functions are called *pure functions*, which we discuss in Section 4.6.

Localizing values: Block

Occasionally, you will need to localize a *value* associated with a symbol without localizing the symbol name itself. For example, you may have a recursive computation that requires you to temporarily reset the system variable `$RecursionLimit`. You can do this with `Block`, thereby only localizing the *value* of `$RecursionLimit` during the evaluation inside the `Block`.

```
In[10]:= Block[{$RecursionLimit = 20},
          x = g[x]
         ]

         $RecursionLimit::reclim :
          Recursion depth of 20 exceeded. More…

Out[10]= g[g[
           g[g[g[g[g[g[g[g[g[g[g[g[g[g[g[g[g[g[g[Hold[g[x]]]]]]]]]]]]]]]]]]]]]
```

Notice the global value of `$RecursionLimit` is unchanged.

```
In[11]:= $RecursionLimit

Out[11]= 256
```

This construct is similar to what is done for the iterators in `Table`, `Do`, `Sum`, and `Product`.

`Module`, on the other hand, would create an entirely new symbol, `$RecursionLimit$`*nn* that would have nothing to do with the global variable `$RecursionLimit`, and so `Module` would be inappropriate for this particular task.

Localizing constants: With

Another scoping construct is available when you simply need to localize constants. If, in the body of your function, you use a variable that is assigned a constant once and never changes, then `With` is the preferred means to localize that constant.

This sets the global variable `y` to have the value 5.

```
In[12]:= y = 5;
```

Here is a simple function that initializes y as a local constant.

```
In[13]:= f[x_] := With[{y = x + 1},
            y
          ]
```

We see the global symbol is unchanged and it does not interfere with the local symbol y inside the With.

```
In[14]:= y
Out[14]= 5

In[15]:= f[2]
Out[15]= 3
```

Using With, you can initialize local constants with the values of global symbols. For example:

```
In[16]:= With[{y = y},
           g[x_] := x + y
         ]
```

This shows that the global value for y was inserted inside g.

```
In[17]:= ?g
Global`g
g[x$_] := x$ + 5
```

Resetting the global value of y has no effect on the localized y inside the With.

```
In[18]:= y = 1;

In[19]:= g[5]
Out[19]= 10
```

Exercises

1. Write a compound function definition for the location of steps taken in an *n*-step random walk on a square lattice. *Hint*: Use the definition for the step increments of the walk as an auxiliary function.
2. The PerfectSearch function defined in Section 1.1 is impractical for checking large numbers because it has to check all numbers from 1 through *n*. If you already

know the perfect numbers below 500, say, it is inefficient to check all numbers from 1 to 1,000 if you are only looking for perfect numbers in the range 500 to 1,000. Modify `searchPerfect` so that it accepts two numbers as input and computes all perfect numbers between the inputs. For example, `PerfectSearch[a,b]` will produce a list of all perfect numbers in the range from *a* to *b*.

3. Overload the `PerfectSearch` function to compute all *3-perfect* numbers. A 3-perfect number is such that the sum of its divisors equals *three* times the number. For example, 120 is 3-perfect since it is equal to three times the sum of its divisors.

```
In[1]:= Apply[Plus, Divisors[120]]
Out[1]= 360
```

Find the only other 3-perfect number under 1,000.
You can overload `PerfectSearch` as defined in Exercise 2 above by defining a three-argument version `PerfectSearch[a,b,3]`.

4. Overload `PerfectSearch` to find the three 4-perfect numbers less than 2,200,000.

5. Redefine `PerfectSearch` so that it accepts as input a number *k*, and two numbers *a* and *b*, and computes all *k*-perfect numbers in the range from *a* to *b*. For example, `PerfectSearch[1,30,2]` would compute all 2-perfect numbers in the range from 1 to 30 and, hence, would output `{6,28}`.

6. If $\sigma(n)$ is defined to be the sum of the divisors of *n*, then *n* is called *superperfect* if $\sigma(\sigma(n)) = 2n$. Write a function `SuperPerfectSearch[a,b]` that finds all superperfect numbers in the range from *a* to *b*.

7. Often in processing files you will be presented with expressions that need to be converted into a format that can be more easily manipulated inside *Mathematica*. For example, a file may contain dates in the form 20030515 to represent May 15 2003. *Mathematica* represents its dates as a list {*year*, *month*, *day*, *hour*, *minutes*, *seconds*}. Write a function `convertToDate[n]` to convert a number consisting of eight digits such as `20030515` into a list of the form `{2003,5,15}`.

```
In[2]:= convertToDate[20030515]
Out[2]= {2003, 5, 15}
```

4.6 Pure functions

A *pure function* is a function that does not have a name and that can be used "on the spot"; that is, at the moment it is created. This is often convenient, especially if the function is only going to be used once or as an argument to a higher-order function, such as `Map`, `Fold`, or `Nest`. The built-in function `Function` is used to create a pure function.

The basic form of a pure function is `Function[`*x*`,`*body*`]` for a pure function with a single variable *x* (any symbol can be used for the variable), and `Function[{`*x*`,`*y*`,...},`*body*`]` for a pure function with more than one variable. The *body* looks like the right-hand side of a user-defined function definition, with the variables *x*, *y*, …, where argument names would be.

As an example, the `square` function we created earlier can be written as a pure function.

```
In[1]:= Function[z, z^2]

Out[1]= Function[z, z^2]
```

There is also a standard input form that can be used in writing a pure function which is easier to write than the `Function` notation but can be a bit cryptic to read. The right-hand side of the function definition is rewritten by replacing the variable by the pound symbol (#) and ending the expression with the ampersand symbol (&) to indicate that this is a pure function.

```
#^2 &
```

If there is more than one variable, `#1`, `#2`, and so on are used.

A pure function can be used exactly like more conventional looking functions, by following the function with the argument values enclosed in square brackets. First we show the pure function using `Function`.

```
In[2]:= Function[z, z^2][6]

Out[2]= 36
```

Here is the same thing, but using the more cryptic shorthand notation (the parentheses in the following example are purely for readability and can be omitted if you wish).

```
In[3]:= (#^2 &)[6]

Out[3]= 36
```

We can, if we wish, give a pure function a name and then use that name to call the function later. This has the same effect as defining the function in the more traditional manner.

```
In[4]:= squared = (#^2) &;
```

```
In[5]:= squared[6]

Out[5]= 36
```

Pure functions are very commonly used with higher-order functions like `Map` and `Apply`, so before going further, let us first look at a few simple examples of the use of pure functions.

Here is a list of numbers.

```
In[6]:= lis = {2, -5, 6.1};
```

Now suppose we wished to square each number and then add 1 to it. The pure function that does this is: `#^2 + 1 &`. So that is what we need to map across this list.

```
In[7]:= Map[#^2 + 1 &, lis]

Out[7]= {5, 26, 38.21}
```

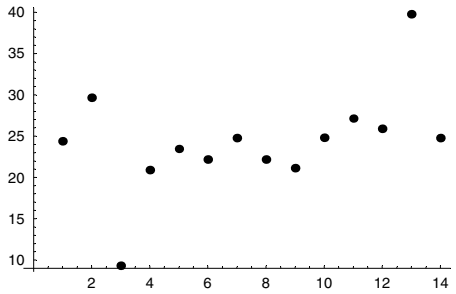
In the next example we will create a set of data and then use the `Select` function to filter out outliers.

```
In[8]:= data = {24.39001, 29.669, 9.321, 20.8856,
          23.4736, 22.1488, 24.7434, 22.1619, 21.1039,
          24.8177, 27.1331, 25.8705, 39.7676, 24.7762}

Out[8]= {24.39, 29.669, 9.321, 20.8856, 23.4736, 22.1488, 24.7434,
         22.1619, 21.1039, 24.8177, 27.1331, 25.8705, 39.7676, 24.7762}
```

A plot of the data shows there are two outliers.

```
In[9]:= ListPlot[data, PlotStyle → PointSize[.02]];
```



The `Select` function takes two arguments — the first is the expression from which it will select elements, and the second argument is a function that must return `True` or `False`. `Select[`*expr*`,`*test*`]` will then select those elements from *expr* that return `True` when *test* is applied to them.

Suppose we wish to exclude all data points that lie outside of the range 20 to 30. Then we need a function that returns `True` if its argument is in that range.

```
In[10]:= Select[data, 20 ≤ # ≤ 30 &]

Out[10]= {24.39, 29.669, 20.8856, 23.4736, 22.1488, 24.7434,
          22.1619, 21.1039, 24.8177, 27.1331, 25.8705, 24.7762}
```

A good way to become comfortable with pure functions is to see them in action, so we will convert some of the functions we defined earlier into pure functions, showing both the `(...#...) &` and the `Function` forms so that you can decide which you prefer to use.

This function tests whether all the elements of a list are even.

```
In[11]:= areEltsEven[lis_] := Apply[And, Map[EvenQ, lis]]

In[12]:= areEltsEven[{2, 4, 5, 8}]

Out[12]= False
```

Here it is written using pure functions.

```
In[13]:= Function[lis, Apply[And, Map[EvenQ, lis]]][{2, 4, 5, 8}]

Out[13]= False

In[14]:= (Apply[And, Map[EvenQ, #1]]) &[{2, 4, 5, 8}]

Out[14]= False
```

This function returns each element in the list greater than all previous elements.

```
In[15]:= maxima[x_] := Union[Rest[FoldList[Max, 0, x]]]

In[16]:= maxima[{2, 6, 3, 7, 9, 2}]

Out[16]= {2, 6, 7, 9}
```

Here it is written using pure functions.

```
In[17]:= Function[x, Union[Rest[FoldList[Max, 0, x]]]][{2, 6, 3, 7, 9, 2}]

Out[17]= {2, 6, 7, 9}

In[18]:= Union[Rest[FoldList[Max, 0, #]]] &[{2, 6, 3, 7, 9, 2}]

Out[18]= {2, 6, 7, 9}
```

We can also create nested pure functions. For example, this maps the pure squaring function over the three-element list `{3,2,7}`.

```
In[19]:= Map[#^2 &, {3, 2, 7}]

Out[19]= {9, 4, 49}
```

When dealing with nested pure functions, the shorthand notation can be used for each of the pure functions but care needs to be taken to avoid confusion as to which # variable belongs to which pure function. This can be avoided by using `Function`, in which case different variable names can be used.

```
In[20]:= Function[y, Map[Function[x, x^2], y]][{3, 2, 7}]

Out[20]= {9, 4, 49}
```

Exercises

1. Write a function to sum the squares of the elements of a numeric list.
2. Write a function to sum the digits of any integer. You will need the `IntegerDigits` function (use `?IntegerDigits`, or look up `IntegerDigits` in the Help Browser to find out about this function).
3. Using the definition of the `distance` function from Exercise 3 of Section 4.4, write a new function `diameter[`*pts*`]` that, given a set of points in the plane, finds the maximum distance between all pairs of points. Try to incorporate the `distance` function into `diameter` without naming it explicitly; that is, use it as a pure function. Consider using `Distribute` to get the set of all pairs of points.

```
In[1]:= pts = {p1, p2, p3};

In[2]:= Distribute[{pts, pts}, List]

Out[2]= {{p1, p1}, {p1, p2}, {p1, p3}, {p2, p1},
         {p2, p2}, {p2, p3}, {p3, p1}, {p3, p2}, {p3, p3}}
```

4. Take the `removeRand` function defined in Section 4.4 and rewrite it as a pure function.

```
In[3]:= removeRand[lis_] :=
        Delete[lis, Random[Integer, {1, Length[lis]}]]
```

5. Convert the `deal` function developed earlier into one that uses pure functions. Use the pure function version of the `removeRand` function from the previous exercise in your new `deal` function definition.
6. Create a function `RepUnit[`*n*`]` that generates integers of length *n* consisting entirely of 1s. For example `RepUnit[7]` should produce 1111111.

7. Create a function `chooseWithoutReplacement[`*lis*`,`*n*`]` that is a generalization of the `deal` function in that it will work with *any* list.

8. Write a pure function that moves a random walker from one location on a square lattice to one of the four adjoining locations with equal probability. For example, starting at `{0,0}`, the function should return either `{0,1}`, `{0,-1}`, `{1,0}` or `{-1,0}` with equal likelihood. Now, use this pure function with `NestList` to generate the list of step locations for an *n*-step random walk starting at `{0,0}`.

9. Create a function `WordsStartingWith[`*lis*`,`*char*`]` that outputs all those words in *lis* that begin with the character *char*. As a sample list, you can use the dictionary.dat file that comes with *Mathematica*.
   Here is a platform-independent path to the dictionary file.

```
In[4]:= wordfile = ToFileName[{$TopDirectory, "Documentation",
          "English", "Demos", "DataFiles"}, "dictionary.dat"]

Out[4]= C:\Program Files\Wolfram Research\Mathematica\5.1\
          Documentation\English\Demos\DataFiles\dictionary.dat
```

   This reads in the file using `ReadList`, specifying the type of data we are reading in as a `Word`.

```
In[5]:= words = ReadList[wordfile, Word];
```

10. Modify Exercise 9 above so that `WordsStartingWith` accepts a string of arbitrary length as its second argument.

11. A naive approach to polynomial arithmetic would require three additions and six multiplies to carry out the arithmetic in the expression $a\,x^3 + b\,x^2 + c\,x + d$. Using Horner's method for fast polynomial multiplication, this expression can be represented as $d + x(c + x(b + a\,x))$, where there are now half as many multiplies. In general, the number of multiplies for an *n*-degree polynomial is given by:

```
In[6]:= Binomial[n + 1, 2]

Out[6]= 1/2 n (1 + n)
```

   This, of course, grows quadratically with *n*, whereas Horner's method grows linearly. Create a function `Horner[`*lis*`,`*var*`]` that implements Horner's method for polynomial multiplication. Here is some sample input and the corresponding output that your function should generate.

```
In[7]:= Horner[{a, b, c, d}, x]
Out[7]= d + x (c + x (b + a x))

In[8]:= Expand[%]
Out[8]= d + c x + b x^2 + a x^3
```

4.7 One-liners

In the simplest version of a user-defined function, there are no value declarations or auxiliary function definitions; the right-hand side is a single nested function call whose arguments are the names of the arguments on the left-hand side, without the blanks. These "one-liners" are fantastically useful and so we will discuss them in the context of three examples, one from electrical engineering (computing Hamming distance), one from ancient history (the Josephus problem), and the last a simple and practical problem (counting change).

Hamming distance

When a code is transmitted over a channel in the presence of noise, errors will often occur. The task of channel coding is to represent the source information in a manner that minimizes the error probability in decoding. *Hamming distance* is used in source coding to represent an information source with the minimum number of symbols. For two lists of binary symbols, the Hamming distance is defined as the number of nonmatching elements and so gives a measure of the how well these two lists match up.

Let us first think about how we might determine if two binary symbols are identical. `SameQ[x,y]` will return `True` if *x* and *y* are identical.

```
In[1]:= {SameQ[0, 0], SameQ[1, 0], SameQ[1, 1]}
Out[1]= {True, False, True}
```

So we need to thread `SameQ` over the two lists of binary numbers

```
In[2]:= MapThread[SameQ, {{1, 0, 0, 1, 1}, {0, 1, 0, 1, 0}}]
Out[2]= {False, False, True, True, False}
```

and then count up the occurrences of `False`.

```
In[3]:= Count[%, False]

Out[3]= 3
```

So a first definition of `HammingDistance` could be accomplished by putting these last two pieces together.

```
In[4]:= HammingDistance[lis1_, lis2_] :
         Count[MapThread[SameQ, {lis1, lis2}], False]

In[5]:= HammingDistance[{1, 0, 0, 1, 1}, {0, 1, 0, 1, 0}]

Out[5]= 3
```

We might try to solve this problem by a more direct approach. Since we are dealing with binary information, we could use some of the logical binary operators built into *Mathematica*.

Here is our transposed list again.

```
In[6]:= lis = Transpose[{{1, 0, 0, 1, 1}, {0, 1, 0, 1, 0}}]

Out[6]= {{1, 0}, {0, 1}, {0, 0}, {1, 1}, {1, 0}}
```

`BitXor[`*x*`,`*y*`]` returns the bitwise XOR of *x* and *y*. So if *x* and *y* can only be among the binary integers 0 or 1, `BitXor` will return 0 whenever they are the same and will return 1 whenever they are different.

```
In[7]:= Apply[BitXor, {{0, 0}, {1, 0}, {1, 1}}, {1}]

Out[7]= {0, 1, 0}
```

Here then is `BitXor` applied to `lis`.

```
In[8]:= Apply[BitXor, lis, {1}]

Out[8]= {1, 1, 0, 0, 1}
```

And here are the number of 1s that occur in that list.

```
In[9]:= Apply[Plus, %]

Out[9]= 3
```

Summing up, our function `HammingDistance2` first pairs up the lists (`Transpose`), then determines which pairs contain different elements (apply `BitXor`), and finally counts up the number of 1s (`Apply[Plus,…]`).

```
In[10]:= HammingDistance2[lis1_, lis2_] := Apply[Plus,
           Apply[BitXor, Transpose[{lis1, lis2}], {1}]
         ]
```

```
In[11]:= HammingDistance2[{1, 0, 0, 1, 1}, {0, 1, 0, 1, 0}]
Out[11]= 3
```

Let us compare the running times of these implementations using a large data set, in this case two lists consisting of one million 0s and 1s.

```
In[12]:= data1 = Table[Random[Integer], {10^6}];
In[13]:= data2 = Table[Random[Integer], {10^6}];
In[14]:= Timing[HammingDistance[data1, data2]]
Out[14]= {1.162 Second, 499801}
In[15]:= Timing[HammingDistance2[data1, data2]]
Out[15]= {1.392 Second, 499801}
```

Although these times do not look too bad, they are in fact too slow for any serious work with signal processing. The exercises ask you to write an implementation of `Hamming Distance` that runs about two orders of magnitude faster than those presented here.

As an aside, the above computations are not a bad check on the built-in random number generator — we would expect that about one half of the paired up lists would contain different elements.

The Josephus problem

Flavius Josephus was a Jewish historian during the Roman–Jewish war of the first century AD. Through his writings comes the following story:

> *The Romans had chased a group of ten Jews into a cave and were about to attack. Rather than die at the hands of their enemy, the group chose to commit suicide one by one. Legend has it though, that they decided to go around their circle of ten individuals and eliminate every other person until only one was left.*

Who was the last to survive? Although a bit macabre, this problem has a definite mathematical interpretation that lends itself well to a functional style of programming. We will start by changing the problem a bit (the importance of rewording a problem can hardly be overstated; the key to most problem-solving resides in turning something we can not work with into something we can work with). We will restate the problem as follows: n people are lined up. The first person is moved to the end of the line, the second person is removed from the line, the third person is moved to the end of the line, and so on until only one person remains in the line.

The statement of the problem indicates that there is a repetitive action, performed over and over again. It involves the use of the `RotateLeft` function (move the person at the front of the line to the back of the line) followed by the use of the `Rest` function (remove the next person from the line).

```
In[16]:= Rest[RotateLeft[#]] &[{a, b, c, d}]

Out[16]= {c, d, a}
```

At this point it is already pretty clear where this computation is headed. We want to take a list and, using the `Nest` function, perform the pure function call (`Rest[RotateLeft[#]) &` on the list until only one element remains. A list of n elements will need $n - 1$ calls. So we can now write the function, to which we give the apt name `survivor`.

```
In[17]:= survivor[lis_] :=
          Nest[Rest[RotateLeft[#]] &, lis, Length[lis] - 1]
```

Trying out the `survivor` function on a list of ten, we see that the fifth position will be the position of the survivor.

```
In[18]:= survivor[Range[10]]

Out[18]= {5}
```

Tracing the applications of `RotateLeft` in this example gives a very clear picture of what is going on. The following form of `TracePrint` shows only the results of the applications of `RotateLeft` that occur during evaluation of the expression `survivor[Range[6]]`.

```
In[19]:= TracePrint[survivor[Range[6]], RotateLeft]

   RotateLeft

   {2, 3, 4, 5, 6, 1}

   RotateLeft

   {4, 5, 6, 1, 3}

   RotateLeft

   {6, 1, 3, 5}

   RotateLeft

   {3, 5, 1}

   RotateLeft

   {1, 5}

Out[19]= {5}
```

Pocket change

As another example, we will write a program to perform an operation most of us do every day: calculating how much change we have in our pocket. Suppose we have the following collection of coins.

```
In[20]:= coins = {p, p, q, n, d, d, p, q, q, p}
Out[20]= {p, p, q, n, d, d, p, q, q, p}
```

Assume p, n, d, and q represent pennies, nickels, dimes, and quarters, respectively. Let us start by using the `Count` function to determine the number of pennies we have.

```
In[21]:= Count[coins, p]
Out[21]= 4
```

This works. So let us do the same thing for all of the coin types.

```
In[22]:= {Count[coins, p], Count[coins, n],
          Count[coins, d], Count[coins, q]}
Out[22]= {4, 1, 2, 3}
```

Looking at this list, it is apparent that there ought to be a more compact way of writing the list. If we `Map` a pure function involving `Count` and `coins` on to the list `{p,n,d,q}`, it should do the job.

```
In[23]:= Map[(Count[coins, #1] &), {p, n, d, q}]
Out[23]= {4, 1, 2, 3}
```

Now that we know how many coins of each type we have, we want to calculate how much change we have. We first do the calculation *manually* to see what we get for an answer (so we will know when our program works).

```
In[24]:= 4 1 + 1 5 + 2 10 + 3 25
Out[24]= 104
```

From the above computation we see that the lists `{4,1,2,3}` and `{1,5,10,25}` are first multiplied together element-wise and then the elements of the result are added. This suggests a few possibilities.

```
In[25]:= Apply[Plus, ({4, 1, 2, 3} {1, 5, 10, 25})]
Out[25]= 104

In[26]:= {4, 1, 2, 3}.{1, 5, 10, 25}
Out[26]= 104
```

Either of these operations are suitable for the job (to coin a phrase, "there's not a penny, nickel, quarter, or dime's worth of difference"). We will write the one-liner using the first method.

```
In[27]:= pocketChange[x_] :=
          Apply[Plus, Map[(Count[x, #] &), {p, n, d, q}] {1, 5, 10, 25}]

In[28]:= pocketChange[coins]

Out[28]= 104
```

Exercises

1. Write a function to compute the Hamming distance of two binary lists (assumed to be of equal length), using `Select` and an appropriate predicate function.

2. All of the implementations of Hamming distance discussed so far are a bit slow for large datasets. You can get a significant speedup in running times by using functions that are optimized for working with numbers (a topic we discuss in detail in Chapter 8). Write an implementation of Hamming distance using the `Total` function and then compare running times with the other versions discussed in this chapter.

3. One of the best ways to learn how to write programs is to practice reading code. We list below a number of one-liner function definitions along with a very brief explanation of what these user-defined functions do and a typical input and output. Deconstruct these programs to see what they do and then reconstruct them as compound functions without any pure functions.

   a. Determine the frequencies with which distinct elements appear in a list.

```
In[1]:= frequencies[lis_] := Map[({#, Count[lis, #]}) &, Union[lis]]

In[2]:= frequencies[{a, a, b, b, b, a, c, c}]

Out[2]= {{a, 3}, {b, 3}, {c, 2}}
```

   b. Divide up a list into parts each of whose lengths are given by the second argument.

```
In[3]:= split1[lis_, parts_] :=
         (Inner[Take[lis, {#1, #2}] &, Drop[#1, -1] + 1,
             Rest[#1], List] &)[FoldList[Plus, 0, parts]]
```

```
In[4]:= split1[Range[10], {2, 5, 0, 3}]

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

This is the same as the previous program, done in a different way.

```
In[5]:= split2[lis_, parts_] :=
          Map[(Take[lis, # + {1, 0}])&,
             Partition[FoldList[Plus, 0, parts], 2, 1]]
```

c. Another game in the Illinois State Lottery is based on choosing *n* numbers, each between 0 and *s* with no duplicates allowed. Write a user-defined function called `lotto` (after the official lottery names of *Little Lotto* and *Big Lotto*) to perform *sampling without replacement* on an arbitrary list. (*Note*: The difference between this function and the function `chooseWithoutReplacement` is that the order of selection is needed here.)

```
In[6]:= lotto1[lis_, n_] := (Flatten[
             Rest[MapThread[Complement, {RotateRight[#], #}, 1]]] &)[
          NestList[Delete[#, Random[Integer, {1, Length[#]}]] &,
           lis, n]]

In[7]:= lotto1[Range[10], 5]

Out[7]= {10, 3, 2, 7, 6}
```

This is the same as the previous program, done in a different way.

```
In[8]:= lotto2[lis_, n_] := Take[Transpose[Sort[
             Transpose[{Table[Random[], {Length[lis]}], lis}]]][[2]], n]
```

As the `split` and `lotto` programs illustrate, user-defined functions can be written in several ways. The choice as to which version of a program to use has to be based on efficiency. A program whose development time was shorter and which runs faster is *better* than a program which took more time to develop and which runs more slowly. Although concise *Mathematica* programs tend to run fastest, when execution speed is a primary concern (when dealing with very large lists) it is a good idea to take various programming approaches and perform `Timing` tests to determine the fastest program.

4. Use the `Timing` function to determine when (in terms of the relative sizes of the list and the number of elements being chosen) it is preferable to use the different versions of the `lotto` function.

5. Rewrite the `pocketChange` function in two different ways — one, using `Dot`, and the other using `Inner`.

6. Make change with quarters, dimes, nickels, and pennies using the fewest coins.

```
In[9]:= makeChange[x_] :=
         Quotient[FoldList[Mod, x, {25, 10, 5}], {25, 10, 5, 1}]

In[10]:= makeChange[119]

Out[10]= {4, 1, 1, 4}
```

7. Write a one-liner to create a list of the step locations of a two-dimensional random walk that is not restricted to a lattice. *Hint*: Each step length must be the same, so the sum of the squares of the x- and y-components of each step should be equal to 1.

8. Write a one-liner version of `convertToDate` as described in Exercise 7 from Section 4.5. Consider the built-in function `FromDigits`.