

Mathematica Programming Fundamentals: Lecture Notes

Richard Gaylord

These notes form the basis of a series of lectures given by the author, in which the fundamental principles underlying *Mathematica*'s programming language are discussed and illustrated with carefully chosen examples. This is not a transcription of those lectures, but the note set was used to create a set of transparencies which Professor Gaylord showed and spoke about during his lectures. These notes formed the basis for both a single 6 hour one-day lecture and a series of four 90 minute lectures, delivered to professionals and to students.

Introduction

In order to use *Mathematica* efficiently, you need to understand how the *Mathematica* programming language works. This tutorial (which has been extensively field-tested in university courses) is intended to provide you with the background that's needed to write your own code.

Note: While this material can't replace the use of The *Mathematica* Book by Wolfram or *Mathematica*'s online help (which give definitions of built-in functions and simple illustrations of their use), it can help to make you more comfortable with the *Mathematica* programming language which often seems obscure, even enigmatic, when first encountered by someone whose programming experience is with one of the traditional procedural languages.

In this note set, the following aspects of the *Mathematica* programming language are emphasized: the nature of expressions, how expressions are evaluated, how pattern-matching works, creating rewrite rules, and using higher-order functions.

■ Summing the elements in a list

Consider the data structure $\{1, 2, 3\}$. How can we add up the elements in the list?

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

```
Out[1]= 6
```

What's going on here?

■ Everything is an expression

Every quantity entered into *Mathematica* is represented internally as an expression. An expression has the form

```
head[arg1, arg2, ..., argn]
```

where the `head` and `argi` can be other expressions.

For example, if we look at two common quantities, a list data structure, $\{a, b, c\}$, and an arithmetic operation, $a+b+c$, they appear to be quite different, but if we use the `FullForm` function to look at their internal representation

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

```
Out[2]//FullForm=
List[a, b, c]
```

```
In[3]:= FullForm[a + b + c]
```

```
Out[3]//FullForm=
Plus[a, b, c]
```

we see that they differ only in their heads.

The use of a common expression structure to represent everything is not merely cosmetic; it allows us to perform some computations quite simply. For example, to add the elements in a list, it is only necessary to change the head of the expression, `List`, to `Plus`. This can be done using the built-in `Apply` function.

```
In[4]:= ? Apply
```

```
Apply[f, expr] or f @@ expr replaces the head of expr by f. Apply[f, expr,
levels spec] replaces heads in parts of expr specified by levelspec. More...
```

```
In[5]:= Trace[Apply[Plus, {1, 2, 3}]]
```

```
Out[5]= {Plus @@ {1, 2, 3}, 1 + 2 + 3, 6}
```

■ Changing a sum into a list

The obvious approach to this task is to do the same sort of thing that we did to add the elements in a list.

```
In[6]:= Apply[List, a + b + c]
```

```
Out[6]= {a, b, c}
```

This works when the list elements are symbols, but it doesn't work for a list of numbers.

```
In[7]:= Apply[List, 1 + 2 + 3]
```

```
Out[7]= 6
```

In order to understand the reason for the different results obtained above, it is necessary to understand how *Mathematica* evaluates expressions. To do that, we first want to distinguish between the various kinds of expressions.

Expressions

■ Kinds of expressions

Non-atomic expressions

Non-atomic expressions have parts which can be extracted from the expression with the `Part` function, and can be replaced with the `ReplacePart` function. For example:

```
In[8]:= Part[{a, 7, c}, 1]
```

```
Out[8]= a
```

```

In[9]:= {a, 7, c}[[0]]
Out[9]= List

In[10]:= Part[a + b + c, 0]
Out[10]= Plus

In[11]:= ReplacePart[{a, 7, c}, e, 2]
Out[11]= {a, e, c}

```

Atomic expressions

Atomic expressions constitute the basic “building blocks” of the *Mathematica* language. There are three kinds of atomic expressions:

1. A symbol, consisting of a letter followed by letters and numbers (eg., *darwin*)
2. Four kinds of numbers:
 - integer numbers (eg., 4)
 - real numbers (eg., 5.201)
 - complex numbers (eg., 3+4I)
 - rational numbers (eg., 5/7)
3. A string, comprised of letters, numbers and spaces (ie., ASCII characters) between quotes (eg., "Computer Simulations with *Mathematica*")

Atomic expressions differ from non-atomic expressions in several ways:

The `FullForm` of an atomic expression is the atom itself.

```

In[12]:= {FullForm[darwin], FullForm[4], FullForm["Computer Simulations with Mathematica"]}
Out[12]= {darwin, 4, "Computer Simulations with Mathematica"}

```

The head (or 0th part) of an atom is the type of atom that it is.

```

In[13]:= {Head[List], Head["Computer Simulations with Mathematica"], 5[[0]]}
Out[13]= {Symbol, String, Integer}

```

An atomic expression has no parts which can be extracted or replaced.

```

In[14]:= Part["Computer Simulations with Mathematica", 1]
Part::partd : Part specification
Computer Simulations with Mathematica[[1]] is longer than depth of object. More...
Out[14]= Computer Simulations with Mathematica[[1]]

```

Compound expressions

A `CompoundExpression` is an expression consisting of a sequence of expressions separated by semi-colons (;).

`expr1; expr2; ...; exprn`

```
In[15]:= a = 5 + 3; 4 a
```

```
Out[15]= 32
```

Entering an expression

When an expression is entered in *Mathematica*, it is evaluated and the result is returned, unless it is followed by a semi-colon.

```
In[16]:= 4^3
```

```
Out[16]= 64
```

When an expression is followed by a semi-colon, the expression is also evaluated, even though nothing is returned.

```
In[17]:= 2 - 6;
```

```
In[18]:= % + 3
```

```
Out[18]= -1
```

```
In[19]:= %%
```

```
Out[19]= -4
```

When the entered expression is a compound expression, its contents are evaluated sequentially and the result of the last evaluation is returned.

```
In[20]:= Trace[a = 3 + 5; 4 a]
```

```
Out[20]= {a = 3 + 5; 4 a, {{3 + 5, 8}, a = 8, 8}, {{a, 8}, 4 8, 32}, 32}
```

■ How expressions are evaluated

Mathematica is a term rewriting system (TRS). Whenever an expression is entered, it is evaluated by term rewriting using rewrite rules. These rules consist of two parts: a pattern on the left-hand side and a replacement text on the right-hand side. When the lhs of a rewrite rule is found to pattern-match part of the expression, that part is replaced by the rhs of the rule, after substituting values in the expression which match labelled blanks in the pattern into the rhs of the rule. Evaluation then proceeds by searching for further matching rules until no more are found.

The evaluation process in *Mathematica* can be easily understood with the following analogy:

Think of your experiences with using a handbook of mathematical formulas, such as the integral tables of Gradshteyn and Ryzhik. In order to solve an integral, you consult the handbook which contains formulas consisting of a left-hand side (lhs) and a right-hand side (rhs), separated by an ‘equals’ sign. You look for an integration formula in the handbook whose left-hand side has the same form as your integral.

Note: While no two formulas in the handbook have the identical lhs, there may be several whose lhs have the same form as your integral (eg., one lhs might have specific values in the integration limits of in the integrand, while another lhs has

unspecified (dummy) variables for these quantities). When this happens, you use the formula whose lhs gives the closest fit to your integral.

$$\int_0^1 x^2 dx$$

$$\int x^n dx = \frac{x^{n+1}}{n+1}$$

$$\int x^2 dx = \frac{x^3}{3}$$

$$\int_a^b x^n dx = \frac{b^{n+1} - a^{n+1}}{n+1}$$

$$\int_0^1 x^n dx = \frac{1}{n+1}$$

$$\int_a^b x^2 dx = \frac{b^3 - a^3}{3}$$

$$\int_0^1 x^2 dx = \frac{1}{3}$$

You then replace your integral with the right-hand side of the matching lhs and you substitute the specific values in your integral for the corresponding variable symbols in the rhs.

Finally, you look through the handbook for formulas (eg., trigonometric identities or algebraic manipulation) that can be used to change the answer further.

This depiction provides an excellent description of the *Mathematica* evaluation process.

However, the application of the term rewriting process to a *Mathematica* expression requires a bit more discussion because a *Mathematica* expression consists of parts, a head and zero or more arguments which are themselves expressions.

`expr[expr1, expr2, ..., exprn]`

It is therefore necessary to understand the order in which the various parts of an expression are evaluated by term rewriting.

The implementation of the evaluation procedure is (with a few exceptions) straightforward:

1. If the expression is a number or a string, it isn't changed.
2. If the expression is a symbol, it is rewritten if there is an applicable rewrite rule in the global rule base; otherwise, it is unchanged.
3. If the expression is not a number, string or symbol, its parts are evaluated in a specific order:
 - a. The head of the expression is evaluated.
 - b. The arguments of the expression are evaluated from left to right in order. An exception to this occurs when the head is a symbol with a hold attribute (eg., `HoldFirst`, `HoldRest`, or `HoldAll`), so that some of its arguments are left in their unevaluated forms (unless they, in turn, have the head `Evaluate`). For example, the `Set` or `SetDelayed` function which we will discuss in a moment.

4. After the head and arguments of an expression are each completely evaluated, the expression consisting of the evaluated head and arguments is rewritten, after making any necessary changes to the arguments based on the `Attributes` (such as `Flat`, `Listable`, `Orderless`) of the head, if there is an applicable rewrite rule in the global rule base.
5. After carrying out the previous steps, the resulting expression is evaluated in the same way and then the result of that evaluation is evaluated, and so on until there are no more applicable rewrite rules.

The details of the term-rewriting process in steps 2 and 4 are as follows:

- a. part of an expression is pattern-matched by the lhs of a rewrite rule
- b. the values which match labelled blanks in the pattern are substituted into the rhs of the rewrite rule and evaluated.
- c. the pattern-matched part of the expression is replaced with the evaluated result.

With this understanding of the evaluation procedure, we can now understand what happened when we entered

```
In[21]:= Apply[List, 1 + 2 + 3]
Out[21]= 6
```

In evaluating this expression, the argument `1+2+3` was evaluated before the `Apply` function was employed.

Controlling the evaluation

We should point out that the user can (to some extent) wrest control of the evaluation process from *Mathematica* and either force or prevent evaluation. We won't go into the details of doing this but we can indicate functions that can be used for this purpose: `Hold`, `HoldAll`, `HoldFirst`, `HoldRest`, `HoldForm`, `HeldPart`, `ReleaseHold`, `Evaluate`, `Unevaluated`, and `Literal`.

In order to turn the sum into a list, it is necessary to prevent the argument `Plus[1, 2, 3]` from being prematurely evaluated before the symbol `Plus` is replaced with the symbol `List`.

```
In[22]:= Apply[List, Unevaluated[1 + 2 + 3]]
Out[22]= {1, 2, 3}
```

Since term rewriting is based on pattern-matching, we need to look at the various sorts of patterns that *Mathematica* recognizes.

Patterns

■ Blanks

Patterns are defined syntactically, i.e., by the internal representation of an expression as given using `FullForm`.

In general, an expression will be matched by several patterns, of differing specificity. For example, constructing as many patterns to match x^2 , in order of increasing generality.

1. x raised to the power of two.
2. x raised to the power of a number.

3. x raised to the power of something.
4. a symbol raised to the power of two.
5. a symbol raised to the power of a number.
6. a symbol raised to the power of something.
7. something raised to the power of two.
8. something raised to the power of a number.
9. something raised to the power of something.
10. something.

The term ‘something’ used above can be replaced by the term ‘an expression’, so that for example, the last case says that x^2 pattern-matches an expression (which is true since x^2 is an expression). To be precise, we need a notation to designate a pattern that has the form of an expression. We also need to designate a pattern that has the form of a sequence of expressions, consecutive expressions separated by commas.

Patterns are defined in *Mathematica* as expressions that *may* contain blanks. A pattern may contain a single (`_`) blank, a double (`__`) blank, or a triple (`___`) blank (the differences will be discussed shortly).

Note: A pattern can be labelled (given a name) by preceding the blank(s) by a symbol, eg., `name_` or `name__` or `name___`. The labelled pattern is matched by exactly the same expression that matches its unlabeled counterpart (pattern labeling, as we will see, is used to create dummy variables).

Note: A blank can be followed by a symbol, eg., `_h` or `__h` or `___h`, in which case, an expression must have the head `h` to match the pattern (this is used to perform type checking).

■ Pattern-matching an expression

We can use the `MatchQ` function to determine if a particular pattern matches an expression or a sequence of expressions. The most specific pattern-match is between an expression and itself.

```
In[23]:= MatchQ[x^2, x^2]
```

```
Out[23]= True
```

To make more general (less specific) pattern-matches, a single blank is used to represent an individual expression, which can be any data object.

We’ll work with x^2 to demonstrate the use of the `Blank` function in pattern-matching. In the following examples (which are arbitrarily chosen from the many possible pattern matches), we’ll first state the pattern-match and then check it using `MatchQ`.

x^2 pattern matches ‘an expression’.

```
In[24]:= MatchQ[x^2, _]
```

```
Out[24]= True
```

x^2 pattern-matches ‘ x raised to the power of an expression’.

```
In[25]:= MatchQ[x^2, x^_]
```

```
Out[25]= True
```

x^2 pattern-matches ‘ x raised to the power of an integer’ (to put it more formally, ‘ x raised to the power of an expression whose head is `Integer`’).

```
In[26]:= MatchQ[x^2, x^_Integer]
```

```
Out[26]= True
```

x^2 pattern-matches ‘an expression whose head is `Power`’.

```
In[27]:= MatchQ[x^2, _Power]
```

```
Out[27]= True
```

x^2 pattern-matches ‘an expression whose head is a symbol and which is raised to the power 2’.

```
In[28]:= MatchQ[x^2, _Symbol^2]
```

```
Out[28]= True
```

x^2 pattern-matches ‘an expression raised to the power 2’.

```
In[29]:= MatchQ[x^2, _^2]
```

```
Out[29]= True
```

x^2 pattern-matches ‘an expression whose head is a symbol and which is raised to the power of an expression whose head is an integer’ (or stated less formally, ‘a symbol raised to the power of an integer’).

```
In[30]:= MatchQ[x^2, _Symbol^_Integer]
```

```
Out[30]= True
```

x^2 pattern-matches ‘an expression raised to the power of an expression’.

```
In[31]:= MatchQ[x^2, _^_]
```

```
Out[31]= True
```

x^2 pattern-matches ‘ x raised to the power of an expression’ (the label y does not affect the pattern-match).

```
In[32]:= MatchQ[x^2, x^y_]
```

```
Out[32]= True
```

As a final example, we look at

```
In[33]:= MatchQ[5^2, _^_]
```

```
Out[33]= False
```


■ Pattern-matching a sequence of one or more expressions

A sequence consists of a number of expression separated by commas. A double blank represents a sequence of one or more expressions and `___h` represents a sequence of one or more expressions, each of which has head `h`.

For example a sequence in a list pattern-matches a double blank (note: we are pattern-matching the sequence in the list, not the list itself)

```
In[34]:= MatchQ[{a, b, c}, {___}]
```

```
Out[34]= True
```

but the arguments of an empty list (which has no arguments) do not pattern-match a double blank.

```
In[35]:= MatchQ[{}, {___}]
```

```
Out[35]= False
```

An expression that pattern-matches a blank will also pattern match a double blank. For example,

```
In[36]:= MatchQ[x^2, ___]
```

```
Out[36]= True
```

■ Pattern-matching a sequence of zero or more expressions

A triple blank represents a sequence of zero or more expressions and `____h` represents a sequence of zero or more expressions, each of which has the head `h`. For example, the triple blank pattern-matches the empty list.

```
In[37]:= MatchQ[{}, {____}]
```

```
Out[37]= True
```

An expression that pattern-matches a blank and a sequence that pattern-matches a double blank pattern both pattern-match a triple blank pattern.

```
In[38]:= MatchQ[x^2, {____}]
```

```
Out[38]= True
```

It is important to be aware that for the purposes of pattern-matching, a sequence is *not* an expression. For example,

```
In[39]:= MatchQ[{a, b, c}, {_}]
```

```
Out[39]= False
```

■ Alternative pattern-matching

We can make a pattern-match less restrictive by specifying alternative patterns that can be matched.

```
In[40]:= MatchQ[x^2, {_} | _^2]
```

```
Out[40]= True
```

■ Conditional pattern-matching

We can make a pattern-match more restrictive by making it contingent upon meeting certain conditions. (Note: satisfying these conditions will be a necessary, but not sufficient, requirement for a successful pattern-match.)

If the blanks of a pattern are followed by `?test`, where `test` is a predicate (ie., a function that returns a `True` or `False`), then a pattern-match is only possible if `test` returns `True` when applied to the *entire* expression. `?test` is used with built-in predicate functions and with anonymous predicate functions.

```
In[41]:= MatchQ[x^2, _^_?OddQ]
```

```
Out[41]= False
```

```
In[42]:= MatchQ[2, _?(# > 3 &)]
```

```
Out[42]= False
```

```
In[43]:= MatchQ[2, _?(# > 1.5 &)]
```

```
Out[43]= True
```

```
In[44]:= MatchQ[2, _Integer?(# > 3 &)]
```

```
Out[44]= False
```

If part of a labeled pattern is followed by `/; condition`, where `condition` contains labels appearing in the pattern, then a pattern-match is possible only if `condition` returns `True` when applied to the *labelled parts* of an expression. For example,

```
In[45]:= MatchQ[x^2, _^y_]
```

```
Out[45]= True
```

```
In[46]:= MatchQ[a^b, _^y_ /; Head[y] == Symbol]
```

```
Out[46]= True
```

With this understanding of how pattern-matching works in *Mathematica*, we can discuss how to create our own rewrite rules which can be used in term rewriting.

Rewrite rules

■ Built-in functions

Mathematica provides over one thousand built-in functions that can be used for term rewriting. These rules are located in the global rule base whenever *Mathematica* is running. Functions defined in a *Mathematica* package are also placed in the global rule base during the session in which the package is loaded. Functions in the global rule base are always available for term rewriting and they are always used whenever applicable.

■ User-defined functions

In addition to the built-in rewrite rules, user-defined rewrite rules can be created and placed in the global rule base where they are always available, and always used, when applicable for the duration of the ongoing session. However, they are not automatically preserved beyond the session in which they are created.

There are basically two ways to create a user-defined rewrite rule: with the `Set` function and with the `SetDelayed` function.

■ Declaring a value using the `Set (=)` function

A value declaration is essentially a nickname for a value (eg., for a list or number) which can be used in place of the value. It is written using `Set[lhs, rhs]` or, more commonly, as

```
lhs = rhs
```

The lhs starts with a name, starting with a letter followed by letters and/or numbers (with no spaces). The rhs is either an expression or a compound expression enclosed in parentheses.

Note: the name on the lhs may be followed by a set of square brackets containing a sequence of patterns or labelled patterns, and the rhs may contain the labels, without the blanks.

For example, consider the following two simple `Set` functions

```
In[47]:= a = {-1, 1}
```

```
Out[47]= {-1, 1}
```

```
In[48]:= rand1 = Random[Integer, {1, 2}]
```

```
Out[48]= 2
```

Notice that when a `Set` function is entered, a value is returned (unless it is followed by a semi-colon). If we look into the global rule base to see what rewrite rules have been created when `a` and `rand1` were entered

```
In[49]:= ? a
```

```
Global`a
```

```
a = {-1, 1}
```

```
In[50]:= ? rand1
```

```
Global`rand1
```

```
rand1 = 2
```

we find that the rewrite rule associated with `a` is the same as the `Set` function we entered, but the rewrite rule associated with `rand1` differs from the corresponding `Set` function. The reason for this is that **when a `Set` function is entered into the global rule base, its lhs is left unevaluated while its rhs is evaluated** and when the rewrite rule that has been created is used, the unevaluated lhs and the evaluated rhs of the function are used. This property is known as the `HoldFirst` attribute.

```
In[51]:= Attributes[Set]
```

```
Out[51]= {HoldFirst, Protected, SequenceHold}
```

```
In[52]:= ?HoldFirst
```

```
HoldFirst is an attribute which specifies that the first argument
to a function is to be maintained in an unevaluated form. More...
```

The reason for the `Set` function having the `HoldFirst` attribute is easily demonstrated by seeing what happens when `Set` does not have this attribute.

```
In[53]:= Set[a, 6]
```

```
Out[53]= 6
```

```
In[54]:= a
```

```
Out[54]= 6
```

```
In[55]:= ClearAttributes[Set, HoldFirst]
```

```
In[55]:= Attributes[Set]
```

```
Out[55]= {Protected, SequenceHold}
```

```
In[55]:= Set[a, 7]
```

```
Set::setraw : Cannot assign to raw object 6. More...
```

```
Out[55]= 7
```

```
In[55]:= SetAttributes[Set, HoldFirst]
```

```
In[56]:= Attributes[Set]
```

```
Out[56]= {HoldFirst, Protected, SequenceHold}
```

When the rhs is a compound expression enclosed in parentheses, the expressions of the rhs are evaluated in sequence and the rhs of the resulting rewrite rule is the result of the final evaluation. For example,

```
In[57]:= rand2 = (b = {-1, 1}; Random[Real, b])
```

```
Out[57]= -0.642186
```

```
In[58]:= ?rand2
```

```
Global`rand2
```

```
rand2 = -0.642186
```

What happened here is that the `b` was first evaluated to give `{-1, 1}` and this value was then used to evaluate the random number function.

The order of expressions on the rhs is important. An expression on the rhs must appear before it is used in another expression on the rhs. For example,

```
In[59]:= rand3 = (Random[Real, c]; c = {-1, 1})
```

```
Random::randn : Range specification c in
```

```
Random[Real, c] is not a valid number or pair of numbers. More...
```

```
Out[59]= {-1, 1}
```

Note that even though an error message was generated when the first expression in the compound expression was evaluated, the overall evaluation of the compound expression continued by evaluating the second expression and its value was then entered into the global rule base.

```
In[60]:= ? rand3
Global`rand3
rand3 = {-1, 1}
```

```
In[61]:= ? c
Global`c
c = {-1, 1}
```

When a `Set` function is entered, both it and any `Set` or `SetDelayed` functions on the rhs create rewrite rules in the global rule base.

```
In[62]:= ? b
Global`b
b = {-1, 1}
```

```
In[63]:= ? c
Global`c
c = {-1, 1}
```

After a value has been declared by entering a `Set` function, the appearance of the value's name during an evaluation causes the value itself to be substituted in (which is why we say that it acts like a nickname). For example,

```
In[64]:= Abs[rand2]
Out[64]= 0.642186
```

What happened here was that the rewrite rule associated with `rand2` in the global rule base was used as an argument to the `Abs` function.

The lhs of a rewrite rule can only be associated with one value at a time. When a `Set` function is entered, the resulting rewrite rule 'overwrites' any previous rewrite rule with the identical lhs. For example,

```
In[65]:= rand4 = Random[Integer, {1, 2}];
```

```
In[66]:= ? rand4
Global`rand4
rand4 = 2
```

```
In[67]:= rand4 = Random[Integer, {1, 2}];
```

```
In[68]:= ? rand4
Global`rand4
rand4 = 1
```

What we see is that the value of `rand4` was 2 after `rand4` was first entered and this value was then changed to 1 after `rand4` was re-entered.

While the lhs of a rewrite rule can only be associated with one value at a time, a value can be associated with several names, simultaneously. We made use of this earlier when we defined both `b` and `c` as $\{-1, 1\}$.

Finally, user-defined rewrite rules can be removed from the global rule base using either the `Clear` or `Remove` function.

```
In[69]:= Clear[b]

In[70]:= ?b
Global`b

In[71]:= Remove[c]

In[72]:= ?c
Information::notfound : Symbol c not found. More...
```

■ Defining a function using the `SetDelayed (:=)` function

Function definitions (ie., programs) are written as

```
name[arg1_, arg2_, ..., argn_] := (expr1; expr2; ... exprm)
```

The lhs starts with a name. The name is followed by a set of square brackets containing a sequence of labelled patterns, which are symbols ending with one or more underscores (ie., blanks). The rhs is either an expression or a compound expression enclosed in parentheses, containing the labels on the lhs (without the blanks).

For example, consider the function definition

```
f[x_] := Random[Real, {0, x}]
```

We'll enter this program

```
In[73]:= f[x_] := Random[Real, {0, x}]
```

The first thing we notice is that, in contrast to a `Set` function, nothing is returned when a `SetDelayed` function is entered. If we query the rule base,

```
In[74]:= ?f
Global`f

f[x_] := Random[Real, {0, x}]
```

we see that a rewrite rule associated with `f` has been placed in the global rule base that is identical to the `SetDelayed` function. The reason is that **when a `SetDelayed` function is entered both its lhs and the rhs are left unevaluated**. This property is known as the `HoldAll` attribute.

```
In[75]:= Attributes[SetDelayed]
Out[75]= {HoldAll, Protected, SequenceHold}

In[76]:= ?HoldAll
```

```
HoldAll is an attribute which specifies that all arguments
to a function are to be maintained in an unevaluated form. More...
```

A user-defined function is called in the same way as a built-in function is called, by entering its name with specific argument value(s).

```
In[77]:= f[8]
Out[77]= 0.791243
```

Each time the lhs of a SetDelayed rewrite rule is entered with specific argument values, the rhs of the rule is evaluated using these values, and the result is returned.

```
In[78]:= f[8]
Out[78]= 3.20796
```

Note: In contrast to the := function, the = function only evaluates the rhs when it is first entered and thereafter, that same evaluated rhs is returned each time the lhs is entered with specific argument values. For example, consider

```
In[79]:= f[x_] = x;
In[80]:= ? f
Global`f
f[x_] = x
In[81]:= f[9]
Out[81]= 9
In[82]:= f[7]
Out[82]= 7
In[83]:= ? f
Global`f
f[x_] = x
```

The definition of `f` above seems to work fine. However, the problem arises when the rhs of the `Set` function has already had a value assigned to it prior to the entry of the `Set` function.

```
In[84]:= y = 7;
In[85]:= g[y_] = y;
In[86]:= ? g
Global`g
g[y_] = 7
In[87]:= g[3]
Out[87]= 7
```

This problem does not arise when the `SetDelayed` function is used.

```
In[88]:= z = 8;
```

```

In[89]:= g[z_] := z

In[90]:= ?g

Global`g

g[z_] := z

In[91]:= g[2]

Out[91]= 2

```

This property of fresh evaluation of both the lhs and rhs of the `:=` function with each use, is why the `:=` function is used to write programs rather than the `=` function.

When the rhs of the `SetDelayed` function is a compound expression enclosed in parentheses, no rewrite rules are created from the auxiliary functions on the rhs when the function is entered (this is because the rhs is not evaluated). When the program is run (or equivalently, a user-defined function is called) for the first time, all of the auxiliary functions are then placed in the global rule base.

```

In[92]:= g[x_] := (d = 2; x + d)

In[93]:= ?g

Global`g

g[x_] := (d = 2; x + d)

In[94]:= ?d

Global`d

In[95]:= g[3]

Out[95]= 5

In[96]:= ?d

Global`d

d = 2

```

■ Placing constraints on a rewrite rule

The use of a rewrite rule can be restricted by attaching constraints on either the lhs or the rhs of a `SetDelayed` rule. Conditional pattern-matching with `_h` or with `_?` and `_/;` can be attached to the dummy variable arguments on the lhs. Also, `/;` can be placed on the rhs, immediately after the (compound) expression.

```

In[97]:= s[x_?EvenQ] := N[Sqrt[x]]

In[98]:= s[6]

Out[98]= 2.44949

In[99]:= s[5]

Out[99]= s[5]

```


■ Localizing names in a rewrite rule

As we have pointed out, when the lhs of a `Set` or `SetDelayed` function is evaluated (which occurs when a `Set` function is **first entered** and when a `SetDelayed` rewrite rule is **first called**), rewrite rules for all of its auxiliary functions are placed in the global rule base. This can cause a problem if a name being used in a program conflicts with the use of the name elsewhere.

We can prevent a name clash by ‘insulating’ the auxiliary functions within the rewrite rule so that they are not placed in the global rule base as separate rewrite rules; they will only ‘exist’ while being used in the evaluation of the rule.

This is usually done using the `Module` function.

```
lhs := Module[{name1 = val1, name2, ...}, rhs]
```

For example,

```
In[100]:=
  t[y_] := Module[{m}, m = 2; y + m]

In[101]:=
  ? m

Global`m

In[102]:=
  t[3]

Out[102]=
  5

In[103]:=
  ? m

Global`m
```

■ Ordering rewrite rules

When the lhs of more than one built-in and/or user-defined rewrite rule is found to pattern-match an expression (which occurs when the lhs's only differ in their specificity), the choice of which rule to use is determined by the order of precedence:

A user-defined rule is used before a built-in rule.

A more specific rule is used before a more general rule (a rule is more specific, the fewer expression it pattern-matches).

So, for example, if we have two rewrite rules whose lhs's have the same name but whose labelled patterns have different specificity, both rules will appear in the global rule base (since their lhs's are not identical) and the more specific rule will be used in preference to the more general rule. For example, if we enter both of the following function definitions

```
In[104]:=
  f[x_] := x^2
  f[x_Integer] := x^3
```

and then query the rule base,

```

In[106]:=
? f

Global`f

f[x_Integer] := x3

f[x_] := x2

```

Now, entering `f` with a real-valued argument

```

In[107]:=
f[6.]

Out[107]=
36.

```

returns a different result from entering `f` with an integer-valued argument.

```

In[108]:=
f[6]

Out[108]=
216

```

This occurs because while an integer-valued argument pattern-matches both `x_` and `x_Integer` (and hence pattern-matches both of the `f` rewrite rules), the second rule is a more specific pattern-match for the integer value 6.

Note: If *Mathematica* cannot deduce which rule is more general, it uses the rules in the order in which they appear in the global rule base.

The ordering of rewrite rules makes it possible for us to create sets of rewrite rules with the same name that give different results, depending on the arguments used. **This is key to writing rule-based programs.**

Note: It is necessary to be careful about the labelling of patterns in rewrite rules because if two or more rules are identical except for the labelling, these rules will all be placed in the global rule base and it may not be obvious which rule will be used. For example,

```

In[109]:=
w[x_] := x4

In[110]:=
w[_] := Random[]

In[111]:=
w[2]

Out[111]=
16

In[112]:=
? w

Global`w

w[x_] := x4

w[_] := Random[]

```

Transformation rules

There are times when we want a rewrite rule to only be applied to (ie., used inside) a specific expression, rather than being placed in the global rule base where it will be used whenever it pattern-matches an expression. For example, the ‘temporary’ substitution of a value for a name in an expression may be preferable to the permanent assignment of the name to the value via a `Set` function. When this is the case, the `ReplaceAll` function can be used together with a `Rule` or `RuleDelayed` function to create a transformation (or local rewrite) rule which is placed directly after the expression to which it is to be applied.

■ Using the `Rule (->)` function

A `Rule` function is attached to an expression. It is written

$$\text{expression} /. \text{lhs} \rightarrow \text{rhs}$$

The lhs can be written using symbols, numbers or labelled patterns.

When an expression with an attached `Rule` transformation rule is entered, the expression itself is evaluated first. Then, *both* the lhs and the rhs of the `Rule` transformation rule are evaluated. Finally, the fully evaluated transformation rule is used in the evaluated expression. For example

```
In[113]:=
  Clear[a]

In[114]:=
  Table[1, {4}] /. Random[Integer, {0, 1}] -> a

Out[114]=
  {a, a, a, a}

In[115]:=
  Table[1, {4}] /. Random[Integer, {0, 1}] -> a

Out[115]=
  {1, 1, 1, 1}

In[116]:=
  Table[1, {4}] /. Random[Integer, {0, 1}] -> {a, b}[[Random[Integer, {1, 2}]]]

Out[116]=
  {b, b, b, b}

In[117]:=
  Table[1, {4}] /. Random[Integer, {0, 1}] -> {a, b}[[Random[Integer, {1, 2}]]]

Out[117]=
  {1, 1, 1, 1}

In[118]:=
  Table[1, {4}] /. Random[Integer, {0, 1}] -> {a, b}[[Random[Integer, {1, 2}]]]

Out[118]=
  {a, a, a, a}
```

We can attach a list of rules to an expression using

```
expression /. {lhs1 → rhs1, lhs2 → rhs2, ...}
```

For example,

```
In[119]:=
  {a, b, c} /. {c → b, b → a}

Out[119]=
  {a, a, b}
```

Multiple transformation rules are used in parallel. The rules are applied in order so that a later rule in the list is used only if all the earlier rules do not match, and **only one transformation rule at most, is applied to a given part of an expression**, and no matching rules are used thereafter, as the above example illustrates.

■ Using the RuleDelayed (:>) function

A RuleDelayed function is attached to an expression. It is written

```
expression /. lhs :> rhs
```

or, for a list of rules

```
expression /. {lhs1 :> rhs1, lhs2 :> rhs2, ...}
```

The lhs can be written using symbols, numbers or labelled patterns.

When an expression with an attached rule is entered, the expression itself is evaluated first. Then, the lhs of the RuleDelayed transformation rule **is** evaluated but the rhs is **not** evaluated. Finally, the partially evaluated transformation rule is used in the evaluated expression (the unevaluated rhs will be evaluated subsequently).

For example,

```
In[120]:=
  Table[1, {4}] /. Random[Integer, {0, 1}] :> {a, b} [[Random[Integer, {1, 2}]]]

Out[120]=
  {a, b, b, b}

In[121]:=
  Table[1, {4}] /. Random[Integer, {0, 1}] :> {a, b} [[Random[Integer, {1, 2}]]]

Out[121]=
  {b, a, b, a}

In[122]:=
  Table[1, {4}] /. Random[Integer, {0, 1}] :> {a, b} [[Random[Integer, {1, 2}]]]

Out[122]=
  {1, 1, 1, 1}
```

■ Placing constraints on a transformation rule

By placing `/;` condition immediately after a RuleDelayed `:>` transformation rule, its use can be restricted in the same way that using `/;` condition can be used to restrict the use of a SetDelayed rewrite rule.

Note: Placing a `/;` condition after a Rule \rightarrow transformation rule serves no purpose since the rhs of the rule has already been evaluated before it is used and hence the conditional restriction is ignored.

■ Applying a transformation rule repeatedly

To apply one or more transformation rules repeatedly to an expression until the expression no longer changes, the `ReplaceRepeated` function is used. For example,

```
In[123]:=
  {a, b, c} //. {c  $\rightarrow$  b, b  $\rightarrow$  a}

Out[123]=
  {a, a, a}
```

Note: In using `//.` with a list of transformation rules, it is important to keep in mind the order of application of the rules. The transformation rules are not repeatedly applied in order. Rather, each rule, in turn, is applied repeatedly.

rule	evaluated	unevaluated
<code>lhs = rhs</code>	<code>rhs</code>	<code>lhs</code>
<code>lhs := rhs</code>		<code>lhs, rhs</code>
<code>expr /. lhs \rightarrow rhs</code>	<code>expr, lhs, rhs</code>	
<code>expr /. lhs \Rightarrow rhs</code>	<code>expr, lhs</code>	<code>rhs</code>

Functional programming style

Mathematica works with built-in and user-defined functions in ways which are characteristic of the ‘functional’ style of programming.

■ Nested function calls

Consider the following consecutive computations:

```
In[124]:=
  Tan[4.0]

Out[124]=
  1.15782

In[125]:=
  Sin[%]

Out[125]=
  0.915931

In[126]:=
  Cos[%]

Out[126]=
  0.609053
```

We can combine these function calls into a *nested function call*.

```
In[127]:=
  Cos[Sin[Tan[4.0]]]
```

```
Out[127]=
  0.609053
```

Notice that the result of one function call is immediately fed into another function without having to first name (or declare) the result.

A nested function call is the application of a function to the result of applying another function to some argument value. In applying functions successively, it is not necessary to declare the value of the result of one function call prior to using it as an argument in another function call.

We can illustrate the use of nested function calls using a deck of playing cards:

```
In[128]:=
  Range[2, 10]
```

```
Out[128]=
  {2, 3, 4, 5, 6, 7, 8, 9, 10}
```

```
In[129]:=
  Join[%, {J, Q, K, A}]
```

```
Out[129]=
  {2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, A}
```

```
In[130]:=
  Outer[List, {♣, ♦, ♥, ♠}, %]
```

```
Out[130]=
  {{{♣, 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}}}
```

```
In[131]:=
  Flatten[%, 1]
```

```
Out[131]=
  {♣, 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}}
```

Combining these operations, we can define cardDeck by combining the operations above.

```
In[132]:=
  cardDeck = Flatten[Outer[List, {♣, ♦, ♥, ♠}, Join[Range[2, 10], {J, Q, K, A}]], 1]
```

```
Out[132]=
  {♣, 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}}
```

Shuffling a deck of cards

```
In[133]:=
Transpose[Sort[Transpose[{Table[Random[], {52}], cardDeck}]]][[2]]

Out[133]=
{{♠, 6}, {♠, A}, {♥, 9}, {♥, 10}, {♦, 6}, {♥, 7}, {♣, K}, {♦, 4}, {♠, 7}, {♣, 6}, {♠, 9},
{♥, A}, {♥, 3}, {♠, 10}, {♠, J}, {♦, 2}, {♥, 2}, {♠, 5}, {♣, J}, {♣, 4}, {♦, A},
{♠, K}, {♥, 5}, {♠, 3}, {♣, Q}, {♣, A}, {♥, 6}, {♥, Q}, {♠, Q}, {♦, 9}, {♠, 5}, {♠, 3},
{♦, J}, {♠, 9}, {♥, 4}, {♠, 4}, {♥, K}, {♣, 2}, {♦, 10}, {♠, 10}, {♦, Q}, {♦, K},
{♦, 7}, {♦, 5}, {♣, 8}, {♦, 3}, {♥, 8}, {♦, 8}, {♠, 8}, {♠, 2}, {♠, 7}, {♥, J}}
```

Note: We can also shuffle a deck of cards using a transformation rule.

```
In[134]:=
Sort[Transpose[{Table[Random[], {52}], cardDeck}]] /. {_, y_} -> y

Out[134]=
{{♠, K}, {♠, 6}, {♦, 4}, {♣, K}, {♦, A}, {♥, Q}, {♠, 4}, {♠, 7}, {♣, 3}, {♦, 10}, {♠, 10},
{♦, 9}, {♠, 10}, {♦, J}, {♠, 7}, {♦, 3}, {♠, J}, {♥, 2}, {♠, 9}, {♥, A}, {♠, A},
{♥, 6}, {♦, 5}, {♥, 9}, {♠, 3}, {♣, 2}, {♣, A}, {♦, K}, {♠, J}, {♦, 2}, {♥, 8}, {♦, 7},
{♠, 5}, {♠, 6}, {♠, Q}, {♦, 6}, {♥, 10}, {♦, Q}, {♦, 8}, {♠, 2}, {♥, 3}, {♥, J},
{♠, Q}, {♥, 7}, {♥, K}, {♠, 8}, {♠, 8}, {♥, 5}, {♠, 4}, {♥, 4}, {♠, 5}, {♠, 9}}
```

■ Anonymous functions

User-defined *anonymous functions* can be created and used ‘on the spot’ without being named or entered prior to being used.

An anonymous function is written using the same form as the rhs of a rewrite rule, replacing variable symbols with #1, #2, ... and enclosing the expression in parentheses followed by an ampersand (&).

This notation can be demonstrated by converting some simple user-defined functions into anonymous functions. For example, a rewrite rule that squared a value

```
In[135]:=
square[x_] := x^2
```

can be written as an anonymous function and applied to an argument, eg., 5, instantly.

```
In[136]:=
(#^2) &[5]

Out[136]=
25
```

An example of an anonymous function with two arguments, raises the first argument to the power of the second argument.

```
In[137]:=
(#1^#2) &[5, 3]

Out[137]=
125
```

It is important to distinguish between an anonymous function which takes multiple arguments and an anonymous function which takes a list with multiple elements as its argument.

For example, the anonymous function just given doesn’t work with an ordered pair argument.

```
In[138]:=
  (#1^#2) &[{2, 3}]

Function::slotn : Slot number 2 in #1^#2 & cannot be filled from (#1^#2 &)[{2, 3}]. More...

Out[138]=
  {2^#2, 3^#2}
```

If we want to perform the operation on the components of an ordered pair, the appropriate anonymous function is

```
In[139]:=
  (#[[1]]^#[[2]]) &[{2, 3}]

Out[139]=
  8
```

■ Nesting anonymous functions

Anonymous functions can be nested, in which case it is sometimes necessary to use the form

```
Function[x, body]

Function[{x, y, ...}, body]
```

rather than the `#• &` form, in order to distinguish between the arguments of the different anonymous functions.

```
In[140]:=
  (#^3) &[(# + 2) &[3]]

Out[140]=
  125

In[141]:=
  Function[y, y^3] [Function[x, x + 2] [3]]

Out[141]=
  125
```

The two forms can also be used together.

```
In[142]:=
  Function[y, y^3] [(# + 2) &[3]]

Out[142]=
  125

In[143]:=
  (#^3) &[Function[x, x + 2] [3]]

Out[143]=
  125
```

Anonymous functions are useful for making predicates and arguments for higher-order functions.

Note: An anonymous predicate function **must** be written using the `#• &` form.

■ Higher-order functions

A *higher-order function* takes a function as an argument and/or returns a function as a result. This is known as ‘treating functions as first-class objects’. We’ll illustrate the use of some of the most important built-in higher order functions.

Apply

```
In[144]:=
? Apply

Apply[f, expr] or f @@ expr replaces the head of expr by f. Apply[f, expr,
  levelspec] replaces heads in parts of expr specified by levelspec. More...
```

We have already seen Apply used to add the elements of a linear list. Given a nested list argument, Apply can be used on the outer list or the interior lists. For example, for a general function, f, and a nested list.

```
In[145]:=
  Apply[f, {{a, b}, {c, d}}]

Out[145]=
  f[{a, b}, {c, d}]

In[146]:=
  Apply[f, {{a, b}, {c, d}}, 2]

Out[146]=
  {f[a, b], f[c, d]}
```

Map

```
In[147]:=
? Map

Map[f, expr] or f /@ expr applies f to each element on the first level in expr.
  Map[f, expr, levelspec] applies f to parts of expr specified by levelspec. More...
```

For a general function, f, and a linear list.

```
In[148]:=
  Map[f, {a, b, c, d}]

Out[148]=
  {f[a], f[b], f[c], f[d]}
```

For a nested list structure, Map can be applied to either the outer list or to the interior lists, or to both. For example, for a general function g:

```
In[149]:=
  Map[g, {{a, b}, {c, d}}]

Out[149]=
  {g[{a, b}], g[{c, d}]}
```

```
In[150]:=
  Map[g, {{a, b}, {c, d}}, {2}]

Out[150]=
  {{g[a], g[b]}, {g[c], g[d]}}
```

MapThread

```
In[151]:=
  ? MapThread

MapThread[f, {{a1, a2, ... }, {b1, b2, ... }, ... }] gives
  {f[a1, b1, ... ], f[a2, b2, ... ], ... }. MapThread[f, {expr1,
  expr2, ... }, n] applies f to the parts of the expr1 at level n. More...
```

For a general function, g, and a nested list.

```
In[152]:=
  MapThread[g, {{a, b, c}, {x, y, z}}]

Out[152]=
  {g[a, x], g[b, y], g[c, z]}

In[153]:=
  MapThread[List, {{a, b, c}, {x, y, z}}]

Out[153]=
  {{a, x}, {b, y}, {c, z}}

In[154]:=
  MapThread[Plus, {{a, b, c}, {x, y, z}}]

Out[154]=
  {a + x, b + y, c + z}
```

NestList and Nest

Nest performs a nested function call, applying the same function repeatedly.

The **Nest** operation applies a function to a value, then applies the function to the result, and then applies the function to that result and then applies... and so on a specified number of times.

```
In[155]:=
  ? NestList

NestList[f, expr, n] gives a list of
  the results of applying f to expr 0 through n times. More...

In[156]:=
  {0.7, Sin[0.7], Sin[Sin[0.7]], Sin[Sin[Sin[0.7]]]}

Out[156]=
  {0.7, 0.644218, 0.600573, 0.565115}
```

```
In[157]:=
  NestList[Sin, 0.7, 3]

Out[157]=
  {0.7, 0.644218, 0.600573, 0.565115}
```

If we are only interested in the final result of the `NestList` operation, we can use the `Nest` function which does not return the intermediate results.

```
In[158]:=
  ?Nest

Nest[f, expr, n] gives an expression with f applied n times to expr. More...

In[159]:=
  Nest[Sin, 0.7, 3]

Out[159]=
  0.565115
```

FixedPointList and FixedPoint

The `Nest` operation does not stop until it has completed a specified number of function applications. There is another function which performs the `Nest` operation, stopping after whichever of the following occurs first: (a) there have been a specified number of function applications, (b) the result stops changing, or (c) some predicate condition is met.

```
In[160]:=
  ?FixedPointList

FixedPointList[f, expr] generates a list giving the results of applying f
repeatedly, starting with expr, until the results no longer change. More...

In[161]:=
  ?FixedPoint

FixedPoint[f, expr] starts with expr, then
applies f repeatedly until the result no longer changes. More...
```

As an example,

```
In[162]:=
  FixedPointList[Sin, 0.7, 5, SameTest -> (#2 < 0.65 &)]

Out[162]=
  {0.7, 0.644218}

In[163]:=
  FixedPointList[Sin, 0.7, 5, SameTest -> ((#1 - #2) < 0.045 &)]

Out[163]=
  {0.7, 0.644218, 0.600573}
```

Note: In these examples, `#1` refers to the next-to-last element in the list being generated and `#2` refers to the last element in the list.

FoldList and Fold

```
In[164]:=
  ?FoldList

FoldList[f, x, {a, b, ...}] gives {x, f[x, a], f[f[x, a], b], ...}. More...
```

```
In[165]:=
  ?Fold

Fold[f, x, list] gives the last element of FoldList[f, x, list]. More...
```

The `Fold` operation takes a function, a value and a list, applies the function to the value, and then applies the function to the result and the first element of the list, and then applies the function to the result and the second element of the list and so on. For example,

```
In[166]:=
  Fold[Plus, 0, {a, b, c, d}]

Out[166]=
  a + b + c + d

In[167]:=
  FoldList[Plus, 0, {a, b, c, d}]

Out[167]=
  {0, a, a + b, a + b + c, a + b + c + d}

In[168]:=
  FoldList[Plus, 0, Table[Random[Integer], {5}]]

Out[168]=
  {0, 1, 1, 2, 2, 2}
```

Exercises

List manipulation

■ Functions

Append
Complement
Dimensions
Drop
First
Flatten
Intersection
Join
Last
Length
Part
Partition
Position
Prepend
Random
Range
ReplacePart
Rest
Reverse
RotateLeft
RotateRight
Sort
Table
Take
Transpose
Union

■ Exercises

Creating lists

Table, Random, Range

Use Table to create the list $\{1,2,3,4,5,6,7,8,9,10\}$. Use Range to create this same list.

Use Table to create the list $\{2,4,6,8,10\}$. Use Range to create this same list.

Use Table and Random to create a 5 by 5 matrix of random real numbers between 0 and 1.

Use Table and Random to create a 5 by 5 matrix of integers between 0 and 9 inclusive.

Use Table to create the list $\{\{0\},\{0,2\},\{0,2,4\},\{0,2,4,6\},\{0,2,4,6,8\}\}$ in two ways.

Create a 10-element list of random 1's and -1's in two ways.

Use Table to create a list equivalent to Array[f,5] and to create a list equivalent to Array[f,{3,4}]

Measuring lists

Length, Dimensions

What is the length of the output of Range[7]? What are its dimensions?

What is the length of the list {{1,2},{2,3},{3,4},{4,5},{5,6},{6,7}}? What are its dimensions?

Element manipulation

Position, Part, ReplacePart, Take, Drop, Rest, Last, First

Where does 2 appear in the list {1,1,2,3,5}? Confirm your guess with Position, and extract the 2 using Part.

Use ReplacePart to replace the 2 in {1,1,2,3,5} with the string "two".

Use ReplacePart to replace the 2 in the list {{1},{2,3},{4,5,6,7}} with the string "two".

Extract the last 4 elements of {5,3,2,1,1} using four different functions: Part, Take, Drop, Rest.

Extract the 2nd through 4th elements of {5,3,2,1,1} using two different functions: Part, Take.

Remove the first and last elements from {5,3,2,1,1} by using Drop twice.

Create {{1},{2,3,4,5,6}} out of Range[6].

Use the lists {a,b,c,d} and {3,2,4,1} to create the list {c,b,d,a} (one step).

Generate a 5 by 5 matrix whose entries are randomly either random integers or random reals. (Hint: how do you generate a number that's randomly either a random integer or a random real?)

Manipulating lists

Reverse, RotateRight, RotateLeft, Join, Prepend, Append, Partition, Transpose, Flatten

In this section, lis represents the matrix {{1,2,3},{4,5,6},{7,8,9}}.

What will Reverse do to lis?

Use RotateRight to change lis into {{7,8,9},{1,2,3},{4,5,6}}. Use RotateLeft to do the same thing.

Use RotateLeft to change lis into {{2,3,1},{5,6,4},{8,9,7}}. Use RotateRight to do the same thing.

Create lis using Range and Partition.

Create {{1,2},{2,3},{3,4},{4,5},{5,6},{6,7}} using Range and Partition. Do the same thing using Table.

Use Range, Partition and Flatten to create the list {1,2,2,3,3,4,4,5,5,6,6,7}.

What will Transpose do to lis?

Use Transpose to collect the x's together and the y's together in the list $\{\{x1,y1\},\{x2,y2\},\{x3,y3\},\{x4,y4\}\}$.

Combine the lists $\{1,2,3,4\}$ and $\{0.1,0.2,0.3,0.4\}$ into one list.

Remove the first and last elements from $\{5,3,2,1,1\}$ by using only Rest and Reverse.

Remove the first and last elements from $\{5,3,2,1,1\}$ using RotateRight and Rest. Do the same thing using RotateLeft and Rest.

Add a random real number to the beginning of the list $\{5,3,2,1,1\}$ using Prepend. Do the same thing using Join.

Add a random real number to the end of the list $\{5,3,2,1,1\}$ using Append. Do the same thing using Join.

Given an n by n matrix, add a border of zeros around it. (Eg., given $\{\{1,2,3\},\{4,5,6\},\{7,8,9\}\}$, return $\{\{0,0,0,0,0\},\{0,1,2,3,0\},\{0,4,5,6,0\},\{0,7,8,9,0\},\{0,0,0,0,0\}\}$.)

Set operations on lists

Sort, Union, Complement, Intersection

For each of the following, use the lists $\{1,2,3,4,5\}$, $\{1,3,5\}$ and $\{2,4,5,6\}$

Combine the lists into a single, sorted list.

What effect will Union have on the three lists?

Find the elements common to these lists.

Find the elements unique to the second list.

Construct a list of ten random integers between 0 and 9 inclusive. Extract a list of which integers appear in this list, and which do not.

Use the lists $\{a,b,c,d\}$ and $\{3,2,4,1\}$ to create the list $\{d,b,a,c\}$ (three steps).

Find all elements of the lists $\{a,b,c,d\}$ and $\{a,b,e,f\}$ that are not common to the two lists.

Extract the 5 largest elements from a list of random numbers.

Pattern matching

■ Functions

```

Alternatives ( | )
Blank ( _ )
BlankSequence ( __ )
Cases
Count
DeleteCases
MatchQ
Pattern
PatternTest ( ? )

```

■ Exercises

Creating patterns

```
Blank, Pattern, PatternTest, Alternatives, MatchQ
```

Find 10 patterns that match $\{\{-1,0\},\{3,5.6,c\}\}$. Verify the match with MatchQ.

Use Alternatives to create a pattern that will match 1, -1, a, or 3.14, but nothing else.

Using patterns

```
Count, Cases, DeleteCases
```

Determine how many 10's there are in a random list of one thousand integers between 1 and 10.

Determine how many sublists of `Table[i+j,{i,1,10},{j,1,3}]` contain the number 5.

Extract the sublists of `Table[i+j,{i,1,10},{j,1,3}]` that contain the number 5.

Remove the sublists of `Table[i+j,{i,1,10},{j,1,3}]` that contain the number 5.

Extract every subexpression of $\{1,\{-1,0\},\{3,5.6,\{c\}\}\}$ that matches `_List`.

Remove every positive number from $\{1,\{-1,0\},\{3,5.6,\{c\}\}\}$.

Function definitions

■ Functions

```

Set ( = )
SetDelayed ( := )
Module
Condition ( /; )

```


■ Exercises

Without using `Condition`, define a function, `g`, of three arguments that returns 0 if any two of the arguments are 0, and 1 otherwise.

Using `Condition`, define a function, `f`, of three arguments that returns 0 if any of its arguments is 0, and 1 otherwise.

Define a function, `pointLoc`, that takes an ordered pair $\{x,y\}$ as its argument. It returns zero if one or both of x and y is zero, and otherwise returns the number of the quadrant in which the point $\{x,y\}$ appears. (The first quadrant is where x and y are positive, and the numbering continues counter-clockwise. The fourth quadrant is where x is positive and y is negative.)

Rewrite the following function definition without using `Condition`:

```
h[x_, y_] := x - y /; IntegerQ[x];
h[x_, y_] := x[[1]] + y /; Head[x] == List && IntegerQ[First[x]] && y == 1
```

Define a function, `g`, that takes five arguments, and satisfies the following conditions:

1. If exactly three of the last four arguments are 1's, return a 1.
2. If the first argument and exactly two other arguments are 1's, return a 1.
3. In all other cases, return a 0.

Check your function by making sure `step[1,1,0,0,1]`, `step[1,1,0,1,1]` and `step[1,1,0,0,0]` evaluate the way you expect.

Define a function, `findTriplet`, that returns a list of three of its arguments that add up to 100, if three such arguments exist, and returns the string "Impossible" otherwise. (Hint: it can be done in two lines using `BlankNullSequence` and `Condition`.) Test your function on the first 20 prime numbers (`findTriplet[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71]`).

Write a function that randomly selects a specified number of elements from a linear list with duplicates allowed. (This is called sampling with replacement.)

Write a function that randomly selects a specified number of elements from a list with no duplicates allowed. (This is called sampling without replacement.)

Write a function that interleaves the elements of two lists of unequal length. (Let the first list be longer than the second list.)

Transformation rules

■ Functions

```
ReplaceAll ( /. )
Rule ( -> )
RuleDelayed ( :> )
ReplaceRepeated ( //. )
```

■ Exercises

Given a list of random triplets, count the number of zeros in each using transformation rules.

Given a random 5 by 5 table of zeros and ones, use `Rule` and `ReplaceAll` change all zeros to ones and all ones to zeros.

Use rules to replace every sublist of `Table[i+j,{i,1,10},{j,1,3}]` that contains the number 5 with the list `{5,5,5}`.

Use rules and `ReplaceAll` to convert all sublists of `Table[Random[Integer],{5},{5}]` that contain at least three zeros to `{0,0,0,0,0}`.

Use a transformation rule to take a list of elements and return a list of those elements that are greater than all of the preceding elements in the list.

Sum the elements in a list using the repeated application of a transformation rule.

Anonymous and higher-order functions

■ Functions

```
Function (# - &)
Map
MapThread
Apply
Nest
NestList
FixedPoint
FixedPointList
FoldList
```

■ Exercises

Pure functions

Create a pure function equivalent to the right-hand side of `f[z_]:=3.75*z*(1-z)`. Calculate the value of the function at 0.5.

Create a pure predicate function that returns `True` if its (integer) argument is a perfect square (whose square root is an integer). Use the function as an argument to `Select` to return all the perfect squares in the list `{1,2,3,...,1000}`.

Create a pure function that takes as an argument a list of two numbers, and returns the first number raised to the power of the second. Test it on the list `{2,10}`. Do this two ways: one using the `Part` function and one without using the `Part` function.

Nested function calls

Use a nested function call to take the lists `{3,2,4,1}` and `{a,b,c,d}` and return `{d,b,a,c}`.

Use a nested function call to create a deck of cards and perform a perfect shuffle on it. (A perfect shuffle consists of cutting the deck in half and interleaving the two halves.)

Map, MapThread, and Apply

Rotate each sublist of `{{1,2},{3,4,5},{6,7,8,9}}` one element to the right. Find the length of each sublist.

Given the following “matrix of lists”, extract the second element from each of the innermost lists: `{{0,1,2},{3,4,5},{6,7,8,9}}`

Use `MapThread` to compute the maximum element in each column of a random 5 by 5 array of integers.

Without using Count, find out how many 1s are in a random list of 0s and 1s.

Rotate the matrix $\{\{1,2,3\},\{4,5,6\}\}$ 90 degrees clockwise to form $\{\{4,1\},\{5,2\},\{6,3\}\}$. Use TableForm to show both the original and the final matches.

Using Map and a pure function, add a zero to the beginning and end of each sublist of $\{\{1,2,3\},\{4,5,6\},\{7,8,9\}\}$.

Un-nest the nested lists within the list $\{\{a,a,a\},\{a\},\{\{b,b,b\},\{b,b\}\},\{a,a\}\}$ to $\{a,a,a,a,b,b,b,b,b,b,a,a\}$.

Create a random 5 by 5 matrix of 0s and 1s, and compute the mean of each row of the matrix.

Iteration with Nest, FixedPoint, and Fold

Use Nest to compute $\text{Sin}[\text{Sin}[\text{Sin}[\text{Sin}[\text{Sin}[2.0]]]]]$

Iterate a pure function equivalent to $f[z_]:=3.75*z*(1-z)$ thirty times, starting with $z=0.5$ and saving all the intermediate results.

Create a one-dimensional random walk by iterating (50 times) a function that randomly adds either -1 or 1 to its argument.

Use a nested function call including FoldList, to take a list and return a list of those elements that are greater than all of the preceding elements in the list.

Define a function that takes a list of two random even integers (such as $\{20,2\}$) and adds one to the larger of the two numbers and subtracts one from the smaller. Iterate this function until the two numbers are equal.

Write a function that removes every other element in a list until only one element remains. (This is known as the Josephus problem.)

Write a function that calculates the frequencies of elements in a list. Do this in two ways: without using an anonymous function and using an anonymous function.