Chapter 16. Some Advanced Perl Techniques

What you've seen so far is the core of Perl, the part that you as a Perl user should understand. But there are many other techniques that, while not obligatory, are still valuable tools to have in your toolbox. We've gathered the most important of those for this chapter. This also segues into the continuation of this book, *Intermediate Perl*, which is your next step in Perl.

Don't be misled by the title of the chapter, though; the techniques here aren't especially more difficult to understand than those that you've already seen. They are "advanced" merely in the sense that they aren't necessary for beginners. The first time you read this book, you may want to skip (or skim) this chapter so you can get right to using Perl. Come back to it a month or two later, when you're ready to get even more out of Perl. Consider this entire chapter a huge footnote.

Slices

It often happens that you need to work with only a few elements from a given list. For example, the Bedrock Library keeps information about its patrons in a large file. Each line in the file describes one patron with six colon-separated fields: a person's name, library card number, home address, home phone number, work phone number, and number of items currently checked out. A little bit of the file looks something like this:

```
fred flintstone:2168:301 Cobblestone Way:555-1212:555-2121:3
barney rubble:709918:299 Cobblestone Way:555-3333:555-3438:0
```

One of the library's applications needs only the card numbers and number of items checked out; it doesn't use any of the other data. You could use something like this to get only the fields you need:

```
while (<$fh>) {
  chomp;
  my @items = split /:/;
  my($card_num, $count) = ($items[1], $items[5]);
  ...  # now work with those two variables
}
```

But you don't need the array `@items` for anything else; it seems like a waste. Maybe it would be better for you to assign the result of `split` to a list of scalars, like this:

```
my($name, $card_num, $addr, $home, $work, $count) = split /:/;
```

That avoids the unneeded array `@items` —but now you have four scalar variables that you don't really need. For this situation, some people make up a number of dummy variable names, like `$dummy_1`, that shows they really don't care about that element from the `split`. But Larry thought that was too much trouble, so he added a special use of `undef`. If you use `undef` as an item in a list you're assigning to, Perl simply ignores the corresponding element of the source list:

```
my(undef, $card_num, undef, undef, undef, $count) = split /:/;
```

Is this any better? Well, it has the advantage that you don't use any unneeded variables. But it has the disadvantage that you have to count `undef`s to tell which element is `$count`. And this becomes quite unwieldy if there are more elements in the list. For example, some people who wanted just the `mtime` value from `stat` would write code like this:

```
my(undef, undef, undef, undef, undef, undef, undef,
  undef, undef, $mtime) = stat $some_file;
```

If you use the wrong number of `undef`s, you get the `atime` or `ctime` by mistake, and that's a tough one to debug. There's a better way: Perl can index into a list as if it were an array. This is a *list slice*. Here, since the

`mtime` is item `9` in the list returned by `stat`, you can get it with a subscript:

```
my $mtime = (stat $some_file)[9];
```

NOTE

It's the 10th item, but the index number is `9`, since the first item is at index `0`. This is the same kind of zero-based indexing that we've used already with arrays. The perlfunc documentation helpfully numbers the list for you so you don't have to count them yourself.

Those parentheses are required around the list of items (in this case, the return value from `stat`). If you wrote it like this, it wouldn't work:

```
my $mtime = stat($some_file)[9];  # Syntax error!
```

A list slice has to have a subscript expression in square brackets after a list in parentheses. The parentheses holding the arguments to a function call don't count.

Going back to the Bedrock Library, the list you work with is the return value from `split`. You can now use a slice to pull out item `1` and item `5` with subscripts:

```
my $card_num = (split /:/)[1];
my $count = (split /:/)[5];
```

Using a scalar-context slice like this (pulling just a single element from the list) isn't bad, but it would be more efficient and simpler if you didn't have to do the `split` twice. So let's not do it twice; let's get both values at once by using a list slice in list context:

```
my($card_num, $count) = (split /:/)[1, 5];
```

The indices pull out element `1` and element `5` from the list, returning those as a two-element list. When you assign that to the two `my` variables, you get exactly what we wanted. You do the `slice` just once, and you set the two variables with a simple notation.

A slice is often the simplest way to pull a few items from a list. Here, you can pull just the first and last items from a list, using the fact that index `-1` means the last element:

```
my($first, $last) = (sort @names)[0, -1];
```

This way to get the minimum or maximum from a list is a bit wasteful, but this isn't a chapter about sorting. For a better way, see the functions in the `List::Util` module.

The subscripts of a slice may be in any order and may even repeat values. This example pulls five items from a list of 10:

```
my @names = qw{ zero one two three four five six seven eight nine };
my @numbers = ( @names )[ 9, 0, 2, 1, 0 ];
print "Bedrock @numbers\n";  # says Bedrock nine zero two one zero
```

Array Slice

That previous example could be made even simpler. When slicing elements from an array (as opposed to a list), the parentheses aren't needed. So we could have done the slice like this:

```
my @numbers = @names[ 9, 0, 2, 1, 0 ];
```

This isn't merely a matter of omitting the parentheses; this is actually a different notation for accessing array elements: an *array slice*. In Chapter 3, we said that the at sign on `@names` meant "all of the elements." Actually, in a linguistic sense, it's more like a plural marker, much like the letter "s" in words like "cats" and "dogs." In Perl, the dollar sign means there's just one of something, but the at sign means there's a list of items.

A slice is always a list, so the array slice notation uses an at sign to indicate that. When you see something like `@names[ ... ]` in a Perl program, you need to do just as Perl does and look at the at sign at the beginning as well as the square brackets at the end. The square brackets mean that you're indexing into an array, and the at sign means that you're getting a whole list of elements, not just a single one (which is what the dollar sign would mean). See Figure 16-1.

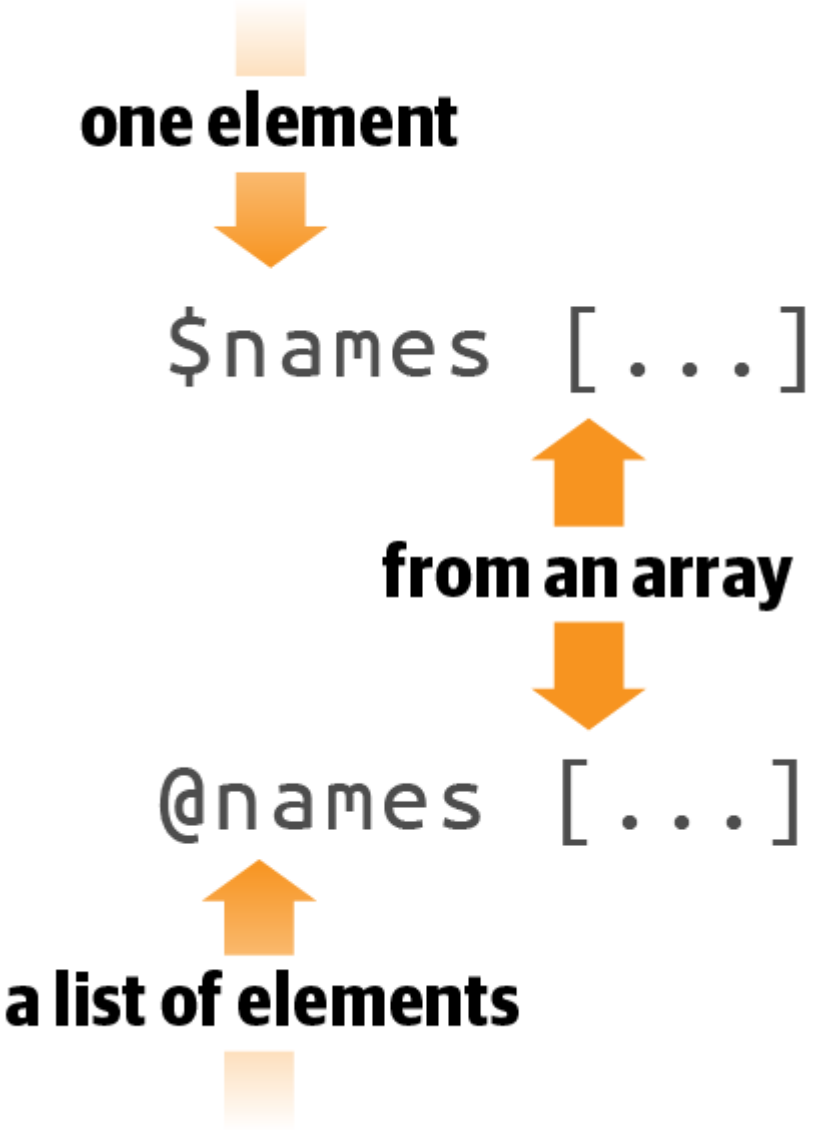

Figure 16-1. Array slices versus single elements

The punctuation mark at the front of the variable reference (either the dollar sign or the at sign) determines the context of the subscript expression. If there's a dollar sign in front, the subscript expression is evaluated in a scalar context to get an index. But if there's an at sign in front, the subscript expression is evaluated in a list context to get a list of indices.

So you see that `@names[ 2, 5 ]` means the same list as `($names[2], $names[5])` does. If you want that list of values, you can simply use the array slice notation. Any place you might want to write the list, you can instead use the simpler array slice.

But you can use the slice in one place where you can't use a list. You can interpolate a slice directly into a string:

```
my @names = qw{ zero one two three four five six seven eight nine };
print "Bedrock @names[ 9, 0, 2, 1, 0 ]\n";
```

If you were to interpolate `@names`, you'd get all of the items from the array, separated by spaces. If instead you interpolate `@names[ 9, 0, 2, 1, 0 ]`, that gives just those items from the array, separated by spaces. Let's go back to the Bedrock Library for a moment. Maybe now your program is updating Mr. Slate's address and phone number in the patron file because he just moved into a large new place in the Hollyrock Hills. If you have a list of information about him in `@items`, you could do something like this to update just those two elements of the array:

```
my $new_home_phone = "555-6099";
my $new_address = "99380 Red Rock West";
@items[2, 3] = ($new_address, $new_home_phone);
```

Once again, the array slice makes a more compact notation for a list of elements. In this case, that last line is the same as an assignment to `($items[2], $items[3])`, but more compact and efficient.

Hash Slice

In a way exactly analogous to an array slice, you can also slice some elements from a hash in a *hash slice*. Remember when three of your characters went bowling, and you kept their bowling scores in the `%score` hash? You could pull those scores with a list of hash elements or with a slice. These two techniques are equivalent, although the second is more concise and efficient:

```
my @three_scores = ($score{"barney"}, $score{"fred"}, $score{"dino"});

my @three_scores = @score{ qw/ barney fred dino/ };
```

A slice is always a list, so the hash slice notation uses an at sign to indicate that. If it sounds as if we're repeating ourselves here, it's because we want to emphasize that hash slices are homologous to array slices. When you see something like `@score{ ... }` in a Perl program, you need to do just as Perl does and look at the at sign at the beginning as well as the curly braces at the end. The curly braces mean that you're indexing into a hash; the at sign means that you're getting a whole list of elements, not just a single one (which is what the dollar sign would mean). See Figure 16-2.

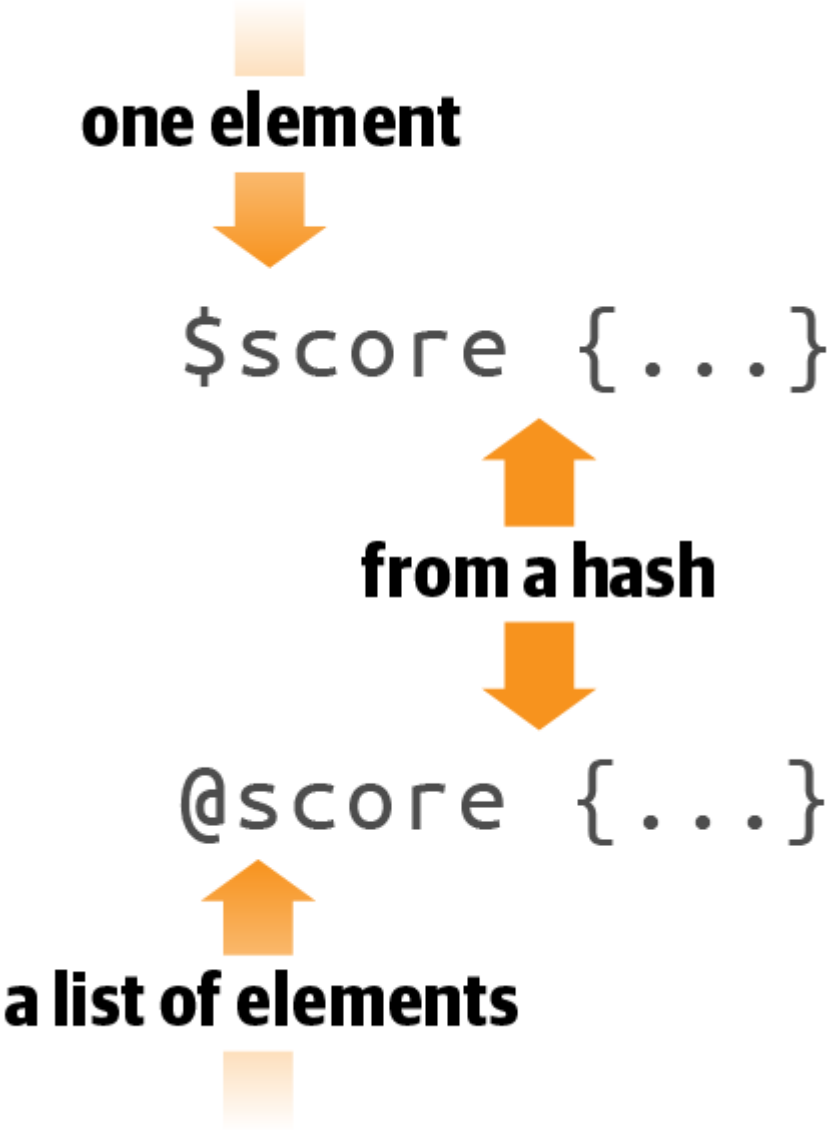

Figure 16-2. Hash slices versus single elements

As you saw with the array slice, the punctuation mark at the front of the variable reference (either the dollar sign or at sign) determines the context of the subscript expression. If there's a dollar sign in front, the subscript expression is evaluated in a scalar context to get a single key. But if there's an at sign in front, the subscript expression is evaluated in a list context to get a list of keys.

It's normal at this point to wonder why there's no percent sign (`%`) here, when we're talking about a hash. That's the marker that means there's a whole hash; a hash slice (like any other slice) is always a *list*, not a hash. In Perl, the dollar sign means there's just one of something, but the at sign means there's a list of items, and the percent sign means there's an entire hash.

As you saw with array slices, a hash slice may be used instead of the corresponding list of elements from the hash, anywhere within Perl. So you can set your friends' bowling scores in the hash (without disturbing any other elements in the hash) in this simple way:

```
my @players = qw/ barney fred dino /;
my @bowling_scores = (195, 205, 30);
@score{ @players } = @bowling_scores;
```

That last line does the same thing as if you had assigned to the three-element list `($score{"barney"}, $score{"fred"}, $score{"dino"})`.

A hash slice may be interpolated too. Here, you print out the scores for your favorite bowlers:

```
print "Tonight's players were: @players\n";
print "Their scores were: @score{@players}\n";
```

Key-Value Slices

Perl v5.20 introduced the *key-value slice* as a way to get out, well, the keys and values that go together. So far, in a hash slice, you got a list of values:

```
my @values = @score{@players};
```

You used an at sign in front of the hash's name because you get out a list of values. After that, `@values` is just the values. If you wanted to remember which keys they went with, you'd have to do extra work:

```
my %new_hash;
@new_hash{ @players } = @values;
```

Or you might try a map (coming up later in this chapter):

```
my %new_hash = map { $_ => $score{$_} } @players;
```

If that's what you want, v5.20 gives you a convenient syntax for it. This time, precede the hash name with a `%`:

```
use v5.20;

my %new_hash = %score{@players};
```

Remember that sigils do not denote variable type; they communicate what you are doing with the variable. In this case, you want key-value pairs. That's a hashy sort of operation, so it gets `%` in front of it.

You can do this with arrays too. Think about the array indices as the keys:

```
my %first_last_scores = %bowling_scores[0,-1];
```

You still use a `%`, because it's still a hashy sort of operation even though it's an array variable. You can tell it's an array because you use the `[ ]` as the subscripting brackets.

Trapping Errors

Sometimes things don't always work out in your programs, but that doesn't mean you want your programs to merely complain before they stop themselves dead. Dealing with errors is a major part of the work of programming, and although we could fill a book on just that, we're still going to give you the introduction. See the third book in this series, *Mastering Perl*, for an in-depth examination of error handling in Perl.

Using eval

Sometimes your ordinary, everyday code can cause a fatal error in your program. Each of these typical statements could crash a program:

```
my $barney = $fred / $dino;      # divide-by-zero error?

my $wilma = '[abc';
print "match\n" if /\A($wilma)/;  # illegal regular expression error?

open my $caveman, '<', $fred     # user-generated error from die?
  or die "Can't open file '$fred' for input: $!";
```

You could go to some trouble to catch some of these, but it's hard to get them all. How could you check the string `$wilma` to ensure it makes a valid regular expression? Fortunately, Perl provides a simple way to catch fatal errors—you can wrap the code in an `eval` block:

```
eval { $barney = $fred / $dino };
```

Now, even if `$dino` is zero, that line won't crash your program. As soon as the `eval` encounters a normally fatal error, it stops the entire block and continues with the rest of the program. Notice that semicolon after the `eval` block. The `eval` is actually an expression (not a control structure, like `while` or `foreach`), so you need that semicolon at the end of the block.

The return value of the `eval` is the last evaluated expression, just like a subroutine. Instead of putting `$barney` on the inside of the `eval`, you

could assign it the result of the `eval`, which allows you to declare `$barney` in the scope outside the `eval`:

```
my $barney = eval { $fred / $dino };
```

If that `eval` catches an error, it returns `undef`. You can use the defined-or operator to set a default value, such as `NaN` ("Not a Number"):

```
use v5.10;
my $barney = eval { $fred / $dino } // 'NaN';
```

When a normally fatal error happens during the execution of an `eval` block, the block is done running, but the program doesn't crash.

When an `eval` finishes, you want to know whether it exited normally or whether it caught a fatal error. If the `eval` caught a fatal error, it returns `undef` and puts the error message in the `$@` special variable, perhaps something like: `Illegal division by zero at my_program line 12`. If there was no error, `$@` will be empty. Of course, that means `$@` is a useful Boolean (true/false) value, true if there was an error. You sometimes see code like this after an `eval` block:

```
use v5.10;
my $barney = eval { $fred / $dino } // 'NaN';
print "I couldn't divide by \$dino: $@" if $@;
```

You can also check the return value, but only if you expect it to be defined if it works. In fact, you should prefer this form to the previous example if it works for your situation:

```
unless( defined eval { $fred / $dino } ) {
    print "I couldn't divide by \$dino: $@" if $@;
}
```

Sometimes the part that you want to test has no meaningful return value even on success, so you can add one yourself. If the `eval` catches a fail-

ure, it won't get the final statement, which is just `1` in this case:

```
unless( eval { some_sub(); 1 } ) {
    print "I couldn't divide by \$dino: $@" if $@;
}
```

In list context, a failed `eval` returns an empty list. In this line, `@averages` only gets two elements if the `eval` fails, because the `eval` doesn't contribute anything to the list:

```
my @averages = ( 2/3, eval { $fred / $dino }, 22/7 );
```

The `eval` block is just like every other Perl block, so it makes a new scope for lexical (`my`) variables and you can have as many statements as you like. Here's an `eval` block hard at work guarding against many potential fatal errors:

```
foreach my $person (qw/ fred wilma betty barney dino pebbles /) {
  eval {
    open my $fh, '<', $person
      or die "Can't open file '$person': $!";

    my($total, $count);

    while (<$fh>) {
      $total += $_;
      $count++;
    }

    my $average = $total/$count;
    print "Average for file $person was $average\n";

    &do_something($person, $average);
  };

  if ($@) {
    print "An error occurred ($@), continuing\n";
  }
}
```

How many possible fatal errors can that `eval` trap? If there is an error in opening the file, you catch it. Calculating the average may divide by zero, but that won't prematurely stop your program. The `eval` even protects the call to the mysteriously named `&do_something` subroutine against fatal errors. This feature is handy if you have to call a subroutine written by someone else, and you don't know whether they've coded defensively enough to avoid crashing your program. Some people purposedly use `die` to signal problems because they expect you to use `eval` to handle it. We'll talk about that more in a moment.

If an error occurs during the processing of one of the files you have in the `foreach` list, you get an error message but your program will go on to the next file without further complaint.

You can also nest `eval` blocks inside other `eval` blocks without Perl getting confused. The inner `eval` traps errors in its block, keeping them from reaching the outer blocks. Of course, after the inner `eval` finishes, if it caught an error you may wish to repost the error by using `die`, thereby letting the outer `eval` catch it. You could change the code to catch an error in the division separately:

```
foreach my $person (qw/ fred wilma betty barney dino pebbles /) {
  eval {
    open my $fh, '<', $person
      or die "Can't open file '$person': $!";

    my($total, $count);

    while (<$fh>) {
      $total += $_;
      $count++;
    }

    my $average = eval { $total/$count } // 'NaN'; # Inner eval
    print "Average for file $person was $average\n";

    &do_something($person, $average);
  };

  if ($@) {
```

```
    print "An error occurred ($@), continuing\n";
  }
}
```

There are four kinds of problems that `eval` can't trap. The first group are syntax errors in the literal source, such as mismatched quotes, missing semicolons, missing operands, or invalid literal regular expressions:

```
eval {
  print "There is a mismatched quote';
  my $sum = 42 +;
  /[abc/
  print "Final output\n";
  };
```

The *perl* compiler catches those errors as it parses the source and stops its work before it starts to run the program. The `eval` can only catch errors once your Perl code is actually running.

The second group are the very serious errors that crash *perl* itself, such as running out of memory or getting an untrapped signal. This sort of error abnormally shuts down the *perl* interpreter itself, and since *perl* isn't running, there's no way it can trap these errors. Some of these errors are listed with an `(X)` code in the perldiag documentation, if you're curious.

The third problem group that an `eval` block can't trap are warnings, either user-generated ones (from `warn`), or Perl's internally generated warnings from the `-w` command-line option or the `use warnings` pragma. There's a separate mechanism apart from `eval` for trapping warnings; see the explanation of the `__WARN__` pseudosignal in the perlvar documentation for the details.

The last sort of error isn't really an error, but this is a good place to note it. The `exit` operator terminates the program at once, even if you call it from a subroutine inside an `eval` block. When you call `exit` , you expect and intend for your program to stop. That's what's supposed to happen, and as such, `eval` doesn't prevent it from doing its work.

We should also mention that there's another form of `eval` that can be dangerous if it's mishandled. In fact, you sometimes run across someone who will say that you shouldn't use `eval` in your code for security reasons. They're (mostly) right that you should use `eval` only with great care, but they're talking about the *other* form of `eval`, sometimes called "`eval` of a string." That `eval` takes a string, compiles it as Perl code, then executes that code just as if you had typed it directly into your program. Notice that the result of any string interpolation has to be valid Perl code:

```
my $operator = 'unlink';
eval "$operator \@files;";
```

If the keyword `eval` comes directly before a block of code in curly braces, as you saw for most of this section, there's no need to worry—that's the safe kind of `eval`.

More Advanced Error Handling

Different languages naturally handle errors in their own way, but a popular concept is the *exception*. You try some code and if anything goes wrong, the program *throws* an exception that it expects you to *catch*. With just basic Perl, you throw an exception with `die` and catch it with `eval`. You can inspect the value of `$@` to figure out what happened:

```
eval {
  ...;
  die "An unexpected exception message" if $unexpected;
  die "Bad denominator" if $dino == 0;
  $barney = $fred / $dino;
  }
if ( $@ =~ /unexpected/ ) {
  ...;
  }
elsif( $@ =~ /denominator/ ) {
  ...;
  }
```

There are many subtle problems with this sort of code, mostly based on the dynamic scope of the `$@` variable. In short, since `$@` is a special variable and your use of `eval` might be wrapped in a higher-level `eval` (even if you don't know about it), you need to ensure that an error you catch doesn't interfere with errors at the higher level:

NOTE

We use `local` here even though we never showed it to you. It replaces a variable's value everywhere in the program until the scope ends. At the end of the scope, the variable has its original value.

```
{
local $@; # don't stomp on higher-level errors

eval {
  ...;
  die "An unexpected exception message" if $unexpected;
  die "Bad denominator" if $dino == 0;
  $barney = $fred / $dino;
  };
if ( $@ =~ /unexpected/ ) {
  ...;
  }
elsif( $@ =~ /denominator/ ) {
  ...;
  }
}
```

That's not the whole story, though, and it's a really tricky problem that's easy to get wrong. The `Try::Tiny` module solves most of this problem for you (and explains it too, if you really need to know). It's not included in the Standard Library, but you can get it from CPAN. The basic form looks like this:

```
use Try::Tiny;

try {
```

```
  ...; # some code that might throw errors
  }
catch {
  ...; # some code to handle the error
  }
finally {
  ...;
  }
```

The `try` acts like the `eval` you just saw. The construct runs the `catch` block only if there was an error. It always runs the `finally` block, allowing you to do any cleanup you'd like to do. You don't need to have the `catch` or the `finally`, either. To simply ignore errors, you can just use the `try`:

```
my $barney = try { $fred / $dino };
```

You can use `catch` to handle the error. Instead of messing with `$@`, `Try::Tiny` puts the error message in `$_`. You can still access `$@`, but part of `Try::Tiny`'s purpose is to prevent the abuse of `$@`:

```
use v5.10;

my $barney =
  try { $fred / $dino }
  catch {
    say "Error was $_"; # not $@
    };
```

The `finally` block runs in either case: if there was an error or not. If it has arguments in `@_`, there was an error:

```
use v5.10;

my $barney =
  try { $fred / $dino }
  catch {
    say "Error was $_"; # not $@
```

```
    }
  finally {
    say @_ ? 'There was an error' : 'Everything worked';
    };
```

Picking Items from a List with grep

Sometimes you want only certain items from a list; maybe it's only the odd numbers from a list of numbers, or maybe it's only the lines mentioning `Fred` from a file of text. As you see in this section, picking some items from a list can be done simply with the `grep` operator.

Try this first one and get the odd numbers from a large list of numbers. You don't need anything new to do that:

```
my @odd_numbers;

foreach (1..1000) {
  push @odd_numbers, $_ if $_ % 2;
}
```

That code uses the modulus operator (`%`), which you saw in Chapter 2. If the number is even, that number "mod two" gives zero, which is false. But an odd number will give one; since that's true, you only `push` the odd numbers onto `@odd_numbers` .

Now, there's nothing wrong with that code as it stands—except that it's a little longer to write and slower to run than it might be, since Perl provides the `grep` operator to act as a filter:

```
my @odd_numbers = grep { $_ % 2 } 1..1000;
```

That line gets a list of 500 odd numbers in one quick line of code. How does it work? The first argument to `grep` is a block that uses `$_` as a placeholder for each item in the list, and returns a Boolean (true/false) value. The remaining arguments are the list of items to search through.

The `grep` operator will evaluate the expression once for each item in the list, much as your original `foreach` loop did. For the ones where the last expression of the block returns a true value, that element is included in the list that results from `grep` .

While the `grep` is running, Perl aliases `$_` to one element of the list after another. You saw this behavior before, in the `foreach` loop. It's generally a bad idea to modify `$_` inside the `grep` expression because this will change the original data too.

The `grep` operator shares its name with a classic Unix utility that picks matching lines from a file by using regular expressions. You can do that with Perl's `grep` , which is much more powerful. Here you select only the lines mentioning `fred` from a file:

```
my @matching_lines = grep { /\bfred\b/i } <$fh>;
```

There's a simpler syntax for `grep` too. If all you need for the selector is a simple expression (rather than a whole block), you can just use that expression, followed by a comma, in place of the block. Here's the simpler way to write that latest example:

```
my @matching_lines = grep /\bfred\b/i, <$fh>;
```

The `grep` operator also has a special scalar context mode in which it can tell you how many items it selected. What if you only wanted to count the matching lines from a file and you didn't care about the lines yourself? You could do that after you created the `@matching_lines` array:

```
my @matching_lines = grep /\bfred\b/i, <$fh>;
my $line_count = @matching_lines;
```

You can skip the intermediate array though (so you don't have to create that array and take up memory) by assigning to the scalar directly:

```
my $line_count = grep /\bfred\b/i, <$fh>;
```

Transforming Items from a List with map

Instead of a filter, you might want to change every item in a list. For example, suppose you have a list of numbers that should be formatted as "money numbers" for output, as with the subroutine `big_money` from Chapter 14. You don't want to modify the original data; you need a modified copy of the list just for output. Here's one way to do that:

```
my @data = (4.75, 1.5, 2, 1234, 6.9456, 12345678.9, 29.95);
my @formatted_data;

foreach (@data) {
  push @formatted_data, big_money($_);
}
```

That looks similar in form to the example code used at the beginning of the previous section on `grep`, doesn't it? So it may not surprise you that the replacement code resembles the first `grep` example:

```
my @data = (4.75, 1.5, 2, 1234, 6.9456, 12345678.9, 29.95);

my @formatted_data = map { big_money($_) } @data;
```

The `map` operator looks much like `grep` because it has the same kind of arguments: a block that uses `$_`, and a list of items to process. And it operates in a similar way, evaluating the block once for each item in the list, with `$_` aliased to a different original list element each time. But `map` uses the last expression of the block differently; instead of giving a Boolean value, the final value actually becomes part of the resulting list. One other important difference is that the expression used by `map` is evaluated in a list context and may return any number of items, not necessarily one each time.

You can rewrite any `grep` or `map` statement as a `foreach` loop pushing items onto a temporary array. But the shorter way is typically more efficient and more convenient. Since the result of `map` or `grep` is a list, it can be passed directly to another function. Here we can print that list of formatted "money numbers" as an indented list under a heading:

```
print "The money numbers are:\n",
  map { sprintf("%25s\n", $_) } @formatted_data;
```

Of course, you could have done that processing all at once, without even the temporary array `@formatted_data` :

```
my @data = (4.75, 1.5, 2, 1234, 6.9456, 12345678.9, 29.95);
print "The money numbers are:\n",
  map { sprintf("%25s\n", big_money($_) ) } @data;
```

As you saw with `grep` , there's also a simpler syntax for `map` . If all you need for the selector is a simple expression (rather than a whole block), you can just use that expression, followed by a comma, in place of the block:

```
print "Some powers of two are:\n",
  map "\t" . ( 2 ** $_ ) . "\n", 0..15;
```

Fancier List Utilities

There are a couple of modules that you can use if you need fancier list handling in Perl. After all, many programs really are just a series of instructions moving lists around in various ways.

The `List::Util` module comes with the Standard Library and provides high-performance versions of common list processing utilities. These are implemented at the C level.

Suppose you wanted to know if a list contains an item that matches some condition. You don't need to get all of the elements, and you want to stop once you find the first matching element. You can't use `grep`, because it always scans the entire list, and if your list is very long, the `grep` might do a lot of extra, unnecessary work:

```
my $first_match;
foreach (@characters) {
  if (/\bPebbles\b/i) {
    $first_match = $_;
    last;
  }
}
```

That's a lot of code. Instead, you can use the `first` subroutine from `List::Util`:

```
use List::Util qw(first);
my $first_match = first { /\bPebbles\b/i } @characters;
```

In the Exercises for Chapter 4, you created the `&total` subroutine. If you knew about `List::Util`, you wouldn't have done so much work:

```
use List::Util qw(sum);
my $total = sum( 1..1000 ); # 500500
```

Also in Chapter 4, the `&max` subroutine did a lot of work to select the largest item from a list. You don't actually need to create that yourself since `List::Util`'s version can do it for you:

```
use List::Util qw(max);
my $max = max( 3, 5, 10, 4, 6 );
```

That `max` deals with numbers only. If you want to do it with strings (using string comparisons), you use `maxstr` instead:

```
use List::Util qw(maxstr);
my $max = maxstr( @strings );
```

If you want to randomize the order of elements in a list, you can use `shuffle`:

```
use List::Util qw(shuffle);
my @shuffled = shuffle(1..1000); # randomized order of elements
```

There's another module, `List::MoreUtils`, that has even more fancy subroutines. This one does not come with Perl, so you need to install it from CPAN. You can check if no, any, or all elements of a list match a condition. Each of these subroutines has the same block syntax of `grep`:

```
use List::MoreUtils qw(none any all);

if (none { $_ < 0 } @numbers) {
  print "No elements less than 0\n"
} elsif (any { $_ > 50 } @numbers) {
  print "Some elements over 50\n";
} elsif (all { $_ < 10 } @numbers) {
  print "All elements are less than 10\n";
}
```

If you want to deal with the list in groups of items, you can use the `natatime` (*N* at a time) to handle that for you:

```
use List::MoreUtils qw(natatime);

my $iterator = natatime 3, @array;
while( my @triad = $iterator->() ) {
  print "Got @triad\n";
}
```

If you need to combine two or more lists, you can use `mesh` to create the large list that interweaves all of the elements, even if the small arrays are not the same length:

```
use List::MoreUtils qw(mesh);

my @abc = 'a' .. 'z';
my @numbers = 1 .. 20;
my @dinosaurs = qw( dino );

my @large_array = mesh @abc, @numbers, @dinosaurs;
```

This takes the first element of `@abc` and makes it the first element of `@large_array`, then takes the first element of `@numbers` to make it the next element of `@large_array`, and then does the same with `@dinosaurs`. It then goes back to `@abc` to get its next element, and so on through all of the elements. The start of the resulting list in `@large_array` is:

```
a 1 dino b 2  c 3 ...
```

In that output, you should notice that there's an empty element between `2` and `c` (so there are two consecutive spaces after `2`). When `mesh` runs out of elements from one of its input arrays, it fills in spots with `undef`. If you had warnings enabled, you'd get several of them.

There are many more useful and interesting subroutines in `List::MoreUtils`. Before you try to re-create what it already does, check its documentation.

Exercises

See "Answers to Chapter 16 Exercises" for answers to these exercises:

1. [30] Make a program that reads a list of strings from a file, one string per line, and then lets the user interactively enter patterns that may match some of the strings. For each pattern, the program should tell how many strings from the file matched, then which ones those were. Don't reread the file for each new pattern; keep the strings in memory. The filename may be hardcoded in the file. If a pattern is invalid (for

example, if it has unmatched parentheses), the program should simply report that error and let the user continue trying patterns. When the user enters a blank line instead of a pattern, the program should quit. (If you need a file full of interesting strings to try matching, try the file *sample_text* in the files you've surely downloaded by now from the O'Reilly website; see the Preface.)

2. [15] Write a program to make a report of the access and modification times (in the epoch time) of the files in the current directory. Use `stat` to get the times, using a list slice to extract the elements. Report your results in three columns, like this:

```
fred.txt          1294145029        1290880566
barney.txt        1294197219        1290810036
betty.txt         1287707076        1274433310
```

3. [15] Modify your answer to Exercise 2 to report the times using the YYYY-MM-DD format. Use a `map` with `localtime` and a slice to turn the epoch times into the date strings that you need. Note the `localtime` documentation about the year and month values it returns. Your report should look like this:

```
fred.txt          2011-10-15        2011-09-28
barney.txt        2011-10-13        2011-08-11
betty.txt         2011-10-15        2010-07-24
```

Support Sign Out

©2022 O'REILLY MEDIA, INC. TERMS OF SERVICE PRIVACY POLICY