Perl 1: Introduction to Perl

Introduction to Perl

Welcome to the O'Reilly School of Technology's (OST) **Introduction to Perl** course. We hope you enjoy the process of learning this practical language with us and then using Perl to resolve all sorts of programming challenges.

Course Objectives

When you complete this course, you will be able to:

- create and run Perl programs using basic scalars, arithmetic, conditional statements, and interpolation.
- implement and manipulate strings, functions, operators, lists, loops, and arrays in Perl.
- format output and list content.
- map data with hashes in Perl.
- perform basic Unix commands.
- aggregate and sort data using subroutines.
- read external files in Perl.

Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take a *user-active* approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you *learn to learn* a new skill.

We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Above all, we want to help you to *learn to learn*. We give you the tools to take control of your own learning experience.

When you complete an OST course, you know the subject matter, *and* you know how to expand your knowledge, so you can handle changes like software and operating system updates.

Here are some tips for using O'Reilly School of Technology courses effectively:

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us that we need to improve our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blew through all of the coursework too quickly.
- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you go completely off the rails.
- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is problem solving. Of course, you can always contact your instructor for hints when you need them.
- **Use all available resources!** In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet, reference books, and online help are all fair game.
- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes! Your instructor will keep you at it until you've mastered the skill. We want you to get that satisfied, "I'm so cool! I did it!" feeling. And you'll have some projects to show off when you're done.

Lesson Format

We'll try out lots of examples in each lesson. We'll have you write code, look at code, and edit existing code. The code will be presented in boxes that will indicate what needs to be done to the code inside.

Whenever you see white boxes like the one below, you'll *type* the contents into the editor window to try the example yourself. The CODE TO TYPE bar on top of the white box contains directions for you to follow:

CODE TO TYPE:

```
White boxes like this contain code for you to try out (type into a file to run).

If you have already written some of the code, new code for you to add looks like this.

If we want you to remove existing code, the code to remove ~~will look like this~~.

We may also include instructive comments that you don't need to type.
```

We may run programs and do some other activities in a terminal session in the operating system or other command-line environment. These will be shown like this:

INTERACTIVE SESSION:

```
The plain black text that we present in these INTERACTIVE boxes is
provided by the system (not for you to type). The commands we want you to type look like this.
```

Code and information presented in a gray OBSERVE box is for you to *inspect* and *absorb*. This information is often color-coded, and followed by text explaining the code in detail:

OBSERVE:

```
Gray "Observe" boxes like this contain information (usually code specifics) for you to
observe.
```

The paragraph(s) that follow may provide addition details on **information** that was highlighted in the Observe box.

We'll also set especially pertinent information apart in "Note" boxes:

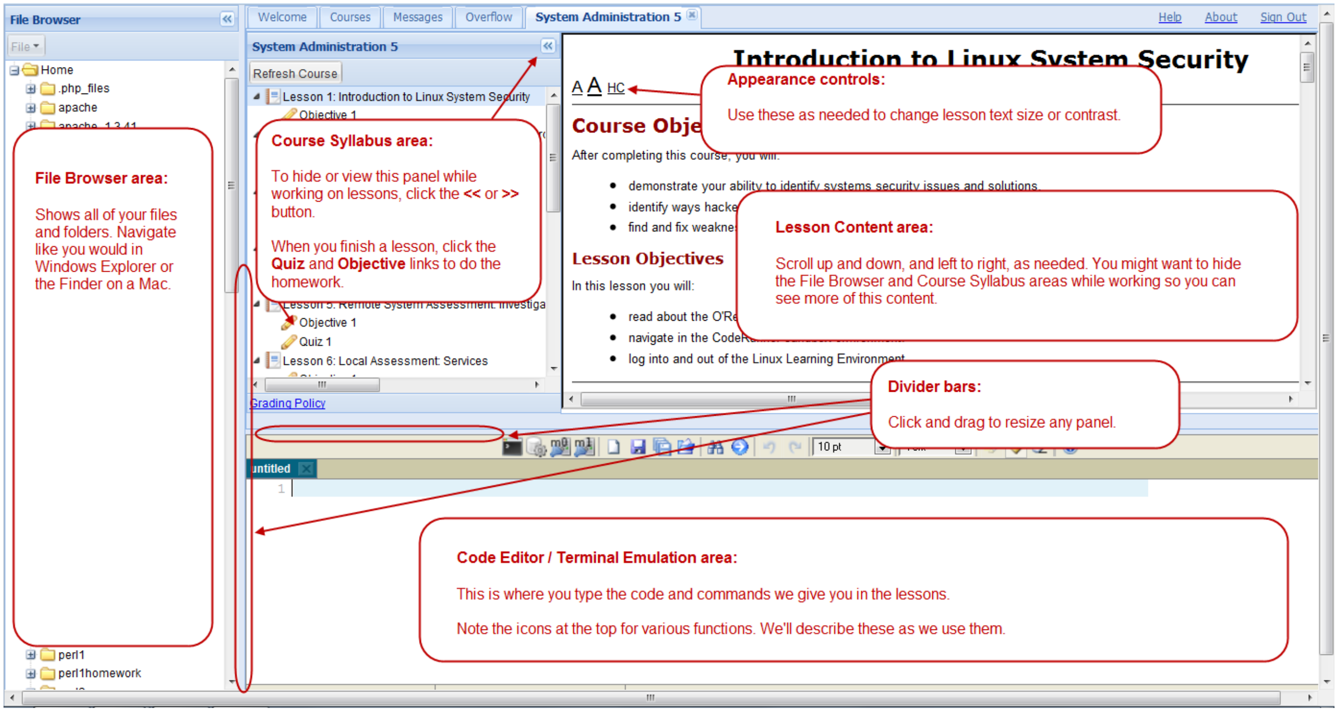
Note Notes provide information that is useful, but not absolutely necessary for performing the tasks at hand.

Tip Tips provide information that might help make the tools easier for you to use, such as shortcut keys.

WARNING Warnings provide information that can help prevent program crashes and data loss.

The CodeRunner Screen

This course is presented in CodeRunner, OST's self-contained environment. We'll discuss the details later, but here's a quick overview of the various areas of the screen:



File Browser area:

Shows all of your files and folders. Navigate like you would in Windows Explorer or the Finder on a Mac.

Course Syllabus area:

To hide or view this panel while working on lessons, click the **<<** or **>>** button.

When you finish a lesson, click the **Quiz** and **Objective** links to do the homework.

Appearance controls:

Use these as needed to change lesson text size or contrast.

Lesson Content area:

Scroll up and down, and left to right, as needed. You might want to hide the File Browser and Course Syllabus areas while working so you can see more of this content.

Divider bars:

Click and drag to resize any panel.

Code Editor / Terminal Emulation area:

This is where you type the code and commands we give you in the lessons.

Note the icons at the top for various functions. We'll describe these as we use them.

These videos explain how to use CodeRunner:

File Management Demo

Code Editor Demo

Coursework Demo

What is Perl?

Perl has been around for a *long* time. It was invented by Larry Wall in 1987. Since then, it has evolved significantly and it's also been the subject of a few myths. Perl has been described as "the duct tape of the internet," because of its adaptability and flexibility. When the Web was invented, people everywhere used Perl to create web pages during the frenzy we now wistfully refer to as *the dot-com boom.*

To give you a bit more background, here are some FAQs and answers about Perl:

- **Q:** Is Perl compiled or interpreted?
- **A:** Perl code is compiled to a kind of byte code, which is then executed, but we don't keep the byte code around between executions. Perl reads source really quickly, so performance isn't an issue. And Perl programs can construct code to be compiled and executed at run time, so the compiler always has to be available. Perl reads all of the code before running any of it, so the code must be syntactically correct before its execution begins.
- **Q:** Is Perl fast?
- **A:** Hand-optimized assembly language will always be fastest, but most of the time, people worry too much about performance—if your program takes 0.1 seconds to run, but it could have been improved to run in 0.01 seconds, most of the time no one will care. We are concerned instead with the speed of development and the ease of maintenance our programs offer. Perl's built-in functions are written in hand-optimized C code, so quite often, a Perl program performing text manipulation will run faster than a casually written C version.
- **Q:** Is Perl difficult to understand?
- **A:** As with any language, people can write code in Perl that's clear or murky. Perl provides multiple ways to execute the same tasks. If programmers learn Perl through copying bad examples, they'll write bad code. But in this course, we'll write Perl that conforms to the best practices and write great code!
- **Q:** What is Perl good for?
- **A:** Just about everything that doesn't involve touching hardware—we don't recommend writing device drivers or operating systems in Perl. Large enterprise-scale applications involving the work of multiple teams need to be developed with discipline and a consistent methodology, because tools that enforce style and interface definitions are uncommon for Perl. But large-scale applications and frameworks have been developed in Perl.
- **Q:** How has Perl been used so far?

- **A:** Some of the most important uses of Perl are secret, because businesses view them as strategic advantages. Some of the known prominent known uses of Perl include: Large parts of the infrastructures of eBay, Yahoo!, and Amazon. Morgan Stanley, Bank of America, Deutsche Bank and other large financial institutions make heavy use of Perl, as do TicketMaster and the British Broadcasting Corporation. Perl use is especially widespread in educational institutions, research agencies such as NASA, and government offices such as the US Courts.
- **Q:** How do I get help using Perl for my project?
- **A:** A great triumph of the Perl community is the Comprehensive Perl Archive Network, or *CPAN*. This is a repository of tens of thousands of downloadable *modules* written by lots of different authors, providing reusable code that performs almost every conceivable task. Using CPAN modules will help you accomplish any Perl task that much faster!

Getting Started

Before we get started with Perl, you need to be familiar with the working environment we'll be using: the Unix environment. Unix allows you to write Perl programs on our computers so you don't need to worry about having any special software on your own computer. We have an entire OST course devoted to Unix if you'd like a deeper understanding of that operating system. For now, let's go over just the elements you'll need to get started learning Perl.

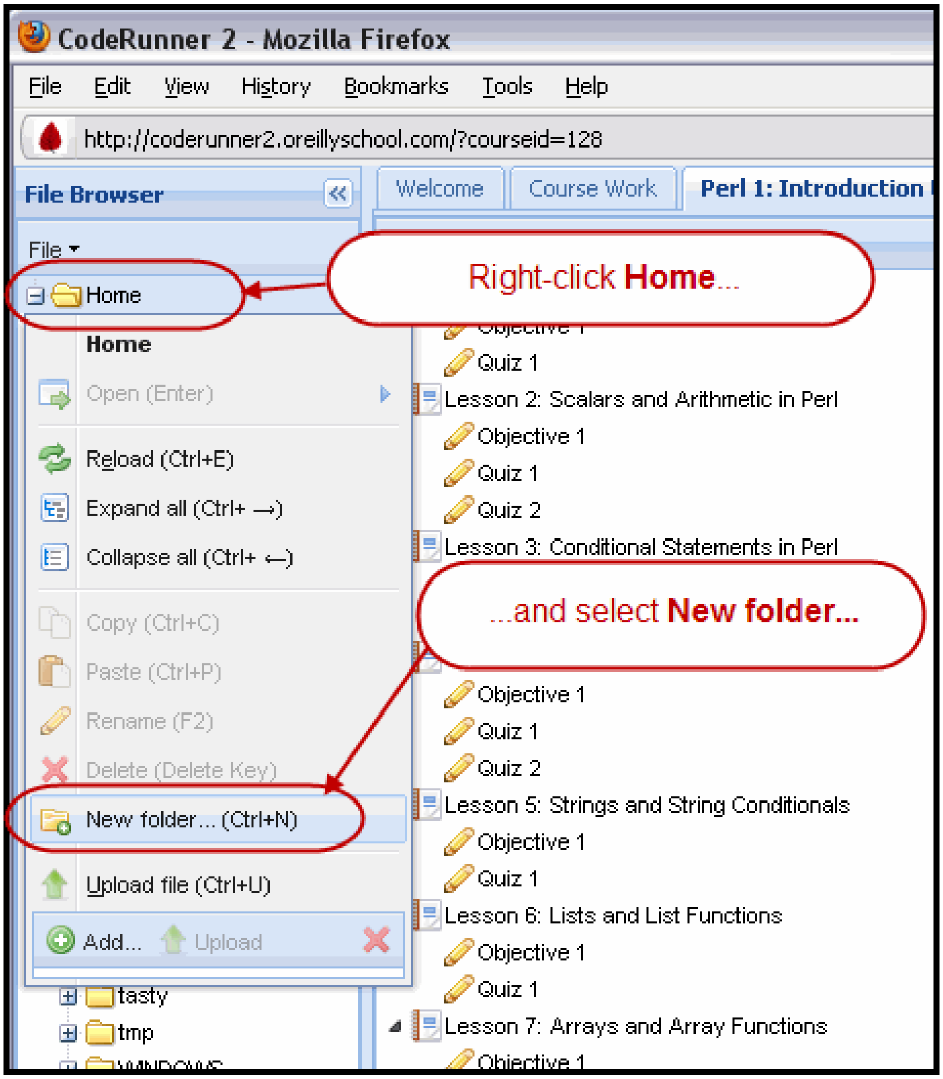
Build and Run a Perl Program

True to OST form, we're going to get to programming right away. Let's make sure you can run a Perl program first.

Usually you'll enter a Perl program in your favorite text editor and then run it by typing a command in a command *shell* (that's the term for one of those special programs whose job in life is to display a prompt and then let you type in a command that causes it to run a program when you press Enter). On a Unix or Linux type of system (that includes OS X), the shell is bash, or tcsh, or some other -sh, and the editor is vim or Emacs or any of many other possibilities. On Windows the shell is the DOS window (what you get by running "cmd") and the editor is Notepad or Wordpad or vim or, again, any of many other possibilities. (You can also run programs on Windows by double-clicking their icons, but then the usual input and output streams are not available.)

In this course we're going to use a completely integrated environment because we want the course to be about learning Perl, not about learning a particular editor and operating system. Later in the course there is a brief lesson on basic Unix commands and editors so that, when you're ready, you can transfer your knowledge of Perl to the real world.

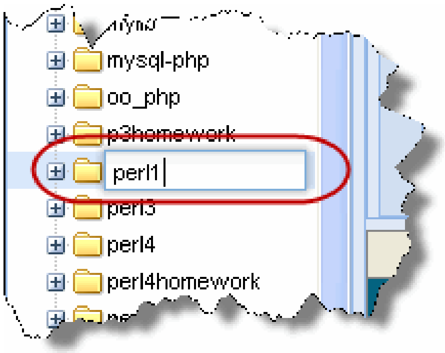
Let's get started on your first program! First, we'll create a folder to keep all of our Perl stuff organized. In the left panel of your CodeRunner window, right-click **Home**, and select **New folder...** as shown:



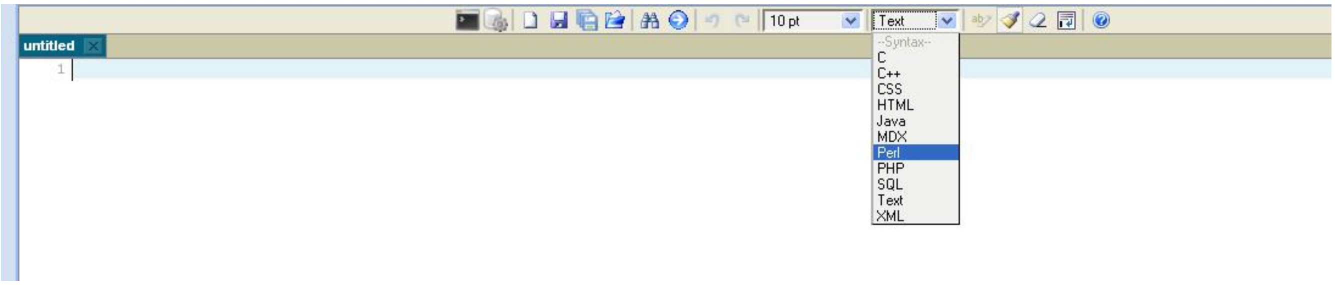
Right-click **Home**...

...and select **New folder...**

Name the new folder **perl1** as shown, and press **Enter**:



Now, select **Perl** in the **Syntax** drop-down menu:



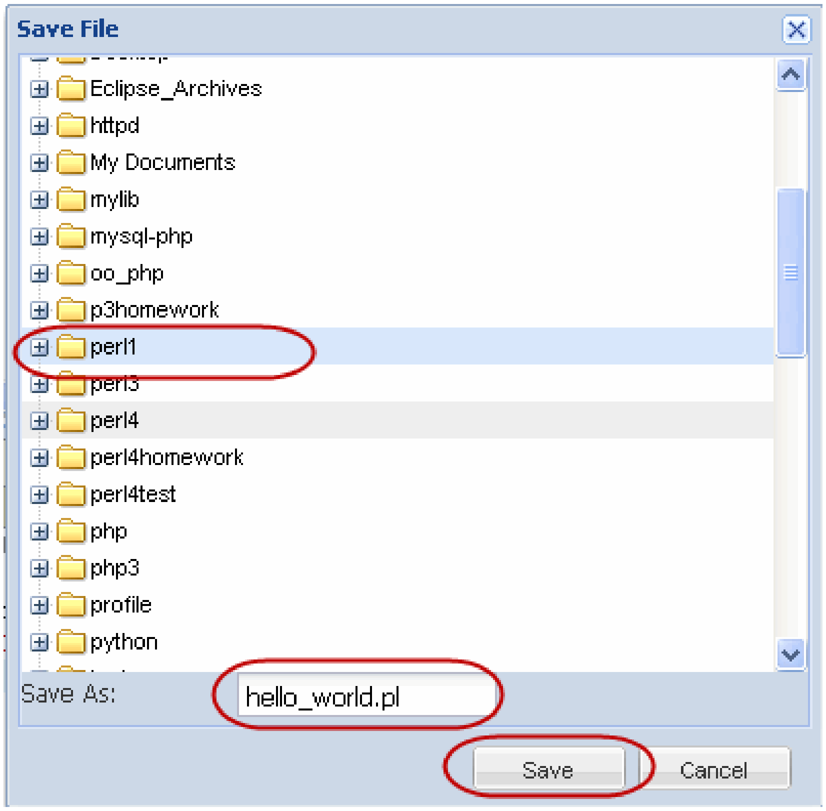
You should now see a Check Syntax icon on the left side of the toolbar.

In the editor, type (don't just cut and paste!) all the lines you see below so it looks like this:

CODE TO TYPE:

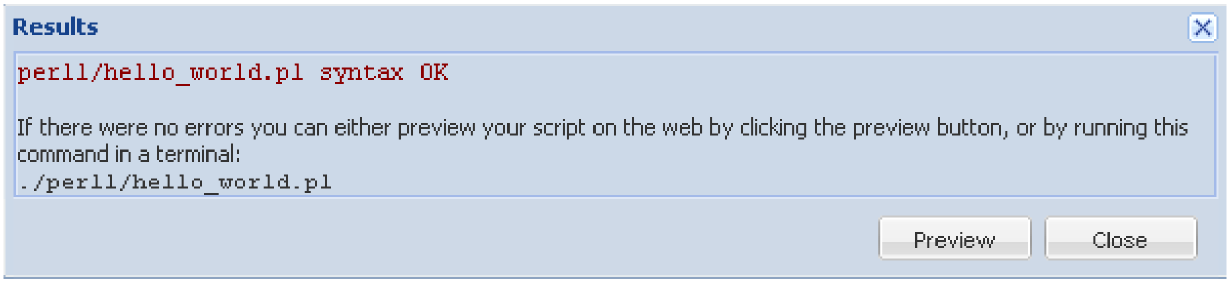
```
#!/usr/bin/perl
use strict;
use warnings;
print "Hello, world!\n";
```

Once you've typed the code, save it and check the syntax. Click the Check Syntax button. A new window will open where you can name and save the file. Select the **/perl1** folder you just created (save all your files here unless otherwise specified), enter the file name **hello_world.pl**, and click **Save**:



Save As: hello_world.pl

You'll see the Results of debugging your program. If everything went alright, you'll see the message **hello_world.pl syntax OK**:



```
perl1/hello_world.pl syntax OK
```

If there were no errors you can either preview your script on the web by clicking the preview button, or by running this command in a terminal:

```
./perl1/hello_world.pl
```

Note that there are two buttons, one labeled **Preview** and one labeled **Close**. The Preview button can be used to view scripts that can be run in a web browser, but in this course we will be writing Perl scripts that can *only be executed on the command line*. Go ahead and click the **Close** button, and let's learn how to do this.

Running Your Program

Now that your program is saved and the syntax checked, let's run it!

Click the **New Terminal** button in the Toolbar: .

Log in to the OST server. Be sure to replace username with your username. It should look something like this:

INTERACTIVE TERMINAL SESSION:

```
login: username
password:
Last login: Mon Aug 16 11:36:41 from 63.171.219.110
cold:~$
```

Note When you enter your password, the cursor will not move. In fact, it will not appear as though anything is happening. Rest assured, it is. Just be sure to type your password carefully and hit Enter. If you are having trouble, please contact your mentor.

The server is named **cold**. All OST students have shells on this server. A **shell** is the place where you execute Unix commands on the server. The commands you execute on your shell will not effect any other shell.

cold:~$ is called a command prompt. If you see this, you're ready to execute Unix commands. To run your program, go to the Unix prompt, and change the directory to the perl1 directory. Then call the program by name:

CODE TO TYPE:

```
cold:~$ cd perl1
cold:~/perl1$ ./hello_world.pl
Hello, world!
cold:~/perl1$
```

Note We're saving all of our programs in the /perl1 folder, so we will change to that folder to run them. As a reminder, we'll show the **cd perl1** command in the interactive session in each lesson.

Congratulations, you've run a Perl program! Sure, it's nothing fancy—it just prints out "Hello World"—but we're off to a good start.

From now on, when we want you to run a program, we'll just say "Check Syntax and run it," and you should follow the procedure above.

You've probably guessed what the **print()** function does, but you may be wondering what those other lines in the program do. The short answer is that every Perl program you write will begin with those lines, so don't worry about them. But we figure that you're the curious type, or you wouldn't be here. So here's the real scoop:

OBSERVE:

```
#!/usr/bin/perl
use strict;
use warnings;
```

The first line is what we call a "shebang" line—all scripts on Unix-like computers start with #! and scripts on Windows mostly ignore that line. What follows the #! characters tells the operating system where to find the program that can read the rest of the file. Now you know that /usr/bin/perl is where to find the specific program that runs Perl programs in this script. The location may vary depending on which computer you're using.

The next two lines of code (in **purple**) enable optional parts of Perl called *pragmas*, that tell Perl to be picky about the code we give it. The result is that Perl will catch and tell us about many common programming errors that we might make.

You will likely encounter Perl programs written by other people who have neglected to include those lines at the beginning. We are not going to be like those programmers. This course uses *best practices* in Perl programming, and using those lines is one of them.

Note An older alternative to "use warnings" is to include a " -w" at the end of the shebang line; if you see that, it's almost as good as "use warnings."

So you're probably wondering, "What happens if I take those lines out?" Let's give it a try and see what happens. Remove the code in ~~red~~ as shown:

CODE TO EDIT:

```
#!/usr/bin/perl
~~use strict;~~
~~use warnings;~~
print "Hello, world!\n";
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./hello_world.pl
Hello, world!
cold:~/perl1$
```

No difference, right? That's because we wrote a good program to begin with. Now let's mess with it a bit. Don't be shy about this, because you're going to mess programs up accidentally anyway; we all do. Try leaving out one of the quotation marks. Edit your code as shown below, removing the ~~red~~ quotation mark:

CODE TO EDIT:

```
#!/usr/bin/perl
print "Hello, world!\n~~"~~;
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./hello_world.pl
Can't find string terminator '"' anywhere before EOF at ./hello_world.pl line 2.
cold:~/perl1$
```

Now, Perl can detect *that* mistake without any help from **use strict** or **use warnings**! Let's try another experiment. Add the blue code as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
print: "Hello, world!\n";
```

Check Syntax and run it. What happens? (We're not going to tell you—you'll just have to try it!)

Okay, now put the two use lines back, keeping the colon in as shown:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;
print: "Hello, world!\n";
```

Check Syntax and run it. Now what happens?

Our Second Program

Let's try a new program that's similar to the first one you created, but with different code. Create a new Perl file in the editor. Select **New File** from the Toolbar: 🗋.

Go ahead and copy the code from hello_world.pl into the new file (we'll let you copy just this once, since you typed it yourself in the first place, but don't get used to it!) and then add the code in blue as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

print "Hello, world!\n";

die "Farewell, cruel world!\n";
```

Check Syntax and name it **hello_world2.pl**. Now run it:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./hello_world2.pl
Hello, world!
Farewell, cruel world!
cold:~/perl1$
```

This program used a new function, **die()**. It might look like **die()** does the same things as **print()**—they both send a string to the screen—but that's not the case. Add the blue code to the program as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

print "Hello, world!\n";

die "Farewell, cruel world!\n";

print "Hello again.\n";
```

Check Syntax and run it:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./hello_world2.pl
Hello, world!
Farewell, cruel world!
cold:~/perl1$
```

Hello again. is not printed this time. You can see why **die** has its name—it causes the program to stop right after it prints its argument.

Scripting

Perl programs are often referred to as scripts, but some people think that means they're not intended for serious work, so we'll call them programs throughout this course. Unlike, say, C or Java programs, Perl programs do not get compiled into a different file that you run. So you never have to wonder whether your source code is the same version as the one you're running—they're the same file!

So is Perl a procedural language, a functional language, or an object-oriented language? Yes, yes, and yes. Perl has elements of all of those types of languages. In this course, we will not be exercising the object-oriented aspect, and functional programming in Perl is usually more of an academic exercise than anything else. We're going to focus our efforts on structured, procedural programming.

Structured means that the program is composed of blocks for handling loops and conditions, instead of executing one command after another or executing commands at random.

Procedural means that the program places sections of code that perform specific functions into special blocks that have names which allow them to be referred to from other places within the code; in Perl these blocks are called *subroutines*. This is also referred to as *modular* programming.

Some languages require you to write only in object-oriented code, as opposed to the conventional model, in which a program is seen as a list of tasks (subroutines) to perform. Perl is not such a language.

Some practices are good to incorporate no matter the language you're using. One of them is *constant refactoring*. Whenever we develop a program, we usually start out with a short list of tasks it needs to execute. When we get the program working, we usually congratulate ourselves, and put it aside. But our work rarely ends there. Usually we'll need to go back and add more capabilities to the program. This is especially true if we did a good job to begin with; people like our work and want more of the same good quality programming to do more for them. Pretty soon a program that was originally designed to perform one little task can grow to do a hundred more. If we don't continually adjust the design of the program to correspond to its expanding purposes, it is likely to become an unwieldy bunch of parts added together without coordination, like an apartment complex grown from a tool shed:

This image provided by Brian Pirie per CC BY 2.0

So as you revise your programs, keep looking for ways to execute the same tasks while improving the design of the code (this is called "refactoring"). One efficient way to refactor is to look for code that is repeated and figure out a way to remove the duplication.

Be sure to make some deliberate errors in your programs during the lessons to see what happens (accidental errors are fine as well). Let's give it a try. Modify your **hello_world2.pl** as shown:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

print "Hello, world!\n"~~;~~

die "Farewell, cruel world!\n";

print "Hello again.\n";
```

Check Syntax and run it to see what happens. Now, modify it again as shown:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

print "Hello, world!\n",

die "Farewell, cruel world!\n";

print "Hello again.\n";
```

Check Syntax and run it to see what happens. Replace the semicolon as shown:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

print "Hello, world!\n";

die "Farewell, cruel world!\n";

print "Hello again.\n";
```

The semicolon is a *statement separator* in Perl; you can put multiple statements on one line or (more commonly) split statements across several lines—Perl doesn't care. Since the end of the line doesn't indicate to Perl where one statement ends and another begins, something else has to—our friend the semicolon. *The shebang line at the top of the program does not count as a statement.*

It isn't crucial that you understand the meaning of an error message entirely, so long as you can locate the error and handle it. But if you do want a more detailed explanation of your error messages, add the code in blue near the top of the program and add the colon again as shown:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;
use diagnostics;
print: "Hello, world!\n";
die "Farewell, cruel world!\n";
print "Hello again.\n";
```

Check Syntax and run it to see what happens.

What do you suppose the two-character sequence \n in the **print()** string does? Try leaving it out, then try adding another one in the middle. Of course, you aren't expected to understand the code completely just yet, but we'll get there.

perldoc -f

There are many built-in functions in Perl, all documented in the giant page that you get with **perldoc perlfunc**. That page is so huge that searching through it for the function you want would result in certain agony. Fortunately, there is an alternative to perldoc that shows you just the part of perlfunc for the function you specify. You get that with the **-f** option to perldoc. Try it in the Unix (or Unix 2) shell. Type the blue code as shown:

CODE TO TYPE:

```
cold:~/perl1$ perldoc -f print
```

You can scroll through the test using the up and down arrows on your keyboard, then when you're ready, exit the page by typing **q**.

You can find the functions listed alphabetically and by category at http://perldoc.perl.org/perlfunc.html.

Phew! That was a long lesson. But you've got a good foundation to build on as we go forward learning Perl. You're doing great so far! Be sure and go back to your syllabus page and complete the Lesson 1 quizzes and projects to make sure you're ready to move on to the next lesson. See you there!

Scalars and Arithmetic

Lesson Objectives

When you complete this lesson, you will be able to:

- write programs to calculate numbers.
- use parentheses to indicate precedence.
- use built-in functions.
- use Perl operators.
- use scalar variables.

Scalars and Arithmetic in Perl

As gratifying as it may be to print strings, we can have a lot more fun using computers to do the "heavy lifting" of calculating numbers. Quick! What's the square root of 147 times 19 plus 32!? We can write a program to help us solve this fast, without ever using a pencil.

Create a new file named **simple_math.pl**, as shown in blue:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

print sqrt(147) * 19 + 32, "\n";
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION

```

 cd perl1
cold:~/perl1$ ./simple_math.pl
262.362757406661
cold:~/perl1$
```

Let's take a look at a few key elements in our code and see what's going on:

- **Commas** The comma indicates that **print()** can take multiple *arguments* (things to print). When we type commas in between the arguments, they'll get printed one after the other. We call that series of things separated by commas a *list*—makes sense, right? (More on those later!)
- **Precision** The answer has a bunch of digits that follow the decimal point. If you know a little about square roots, you know that the real answer would go on forever. So, why doesn't it? Well, you have to stop somewhere. Perl's precision is equal to that of the floating point numbers your computer uses. (You can get it to print more digits if you use a special module called **bignum**; but that's more advanced than we want to be right now.)
- **Parentheses** Why do we have parentheses around the 147? Why *don't* we have them around the arguments to **print**? We can answer these questions by checking out what happens when we change those conditions. First let's take the parentheses away completely:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

print sqrt 147 * 19 + 32, "\n";
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION

```
cold:~/perl1$ ./simple_math.pl
53.1507290636732
cold:~/perl1$
```

Now put the parentheses back in, and add a second pair:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

print(sqrt(147) * 19 + 32, "\n");
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION

```
cold:~/perl1$ ./simple_math.pl
262.362757406661
cold:~/perl1$
```

So what's going on here? First, we need to address a little thing called *precedence*. The original problem—"the square root of 147 times 19 plus 32"—is a bit ambiguous. Does it mean:

- ((the square root of 147) times 19) plus 32
- ((the square root of 147) times (19 plus 32)
- the square root of ((147 times 19) plus 32)

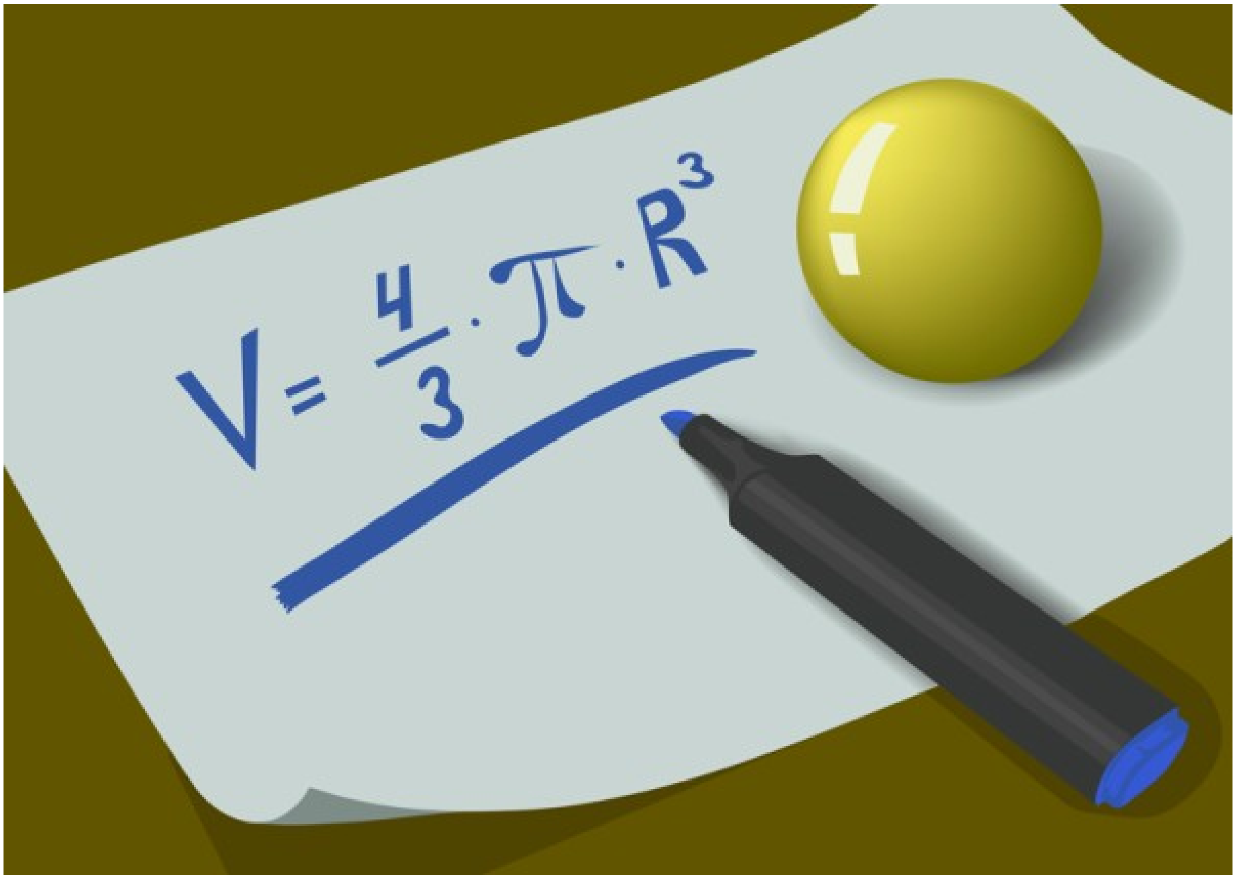
...or any of several other possibilities? I added parentheses so you could figure out the meaning of the expression. We use parentheses the same way in Perl. If you don't have parentheses in place to indicate which tasks to execute first, your Perl program uses rules of *precedence* to determine the order that it will execute operators like **+**, *****, and **sqrt**. So looking back up the page at the results of the totally parentheses-free statement, which parts of the calculation do you suppose Perl did first, second, and third?

Built-in functions like **print()** don't require parentheses around their arguments. Most languages don't give you a choice—you *must* include them—but parentheses are optional in Perl. You can include them if you like, or if you need them to prevent Perl from making the wrong assumption about where your function arguments end. Programmers who are used to working in languages where parentheses are required tend to include them all the time, but with Perl, we usually leave them off the outermost function call (the leftmost one in a statement, for example, **print()** in the lines above). The code is more readable when it isn't cluttered with extra parentheses. In this course we'll always follow the best practices for Perl, including the economical use of parentheses.

Perl's Operators

Perl has *lots* of operators. You can always look up an operator in **perldoc** locally or at:

http://perldoc.perl.org/perlop.html. Let's use some of them to calculate the volume of a sphere with a radius of 7.5 meters:



Create a file named **sphere.pl**. Type the code below into the console as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

print "Volume of sphere of radius 7.5 = ";
print 4/3 * 3.14159265 * (7.5 ** 3), "\n";
```

Check Syntax and run it:

INTERACTIVE TERMINAL SESSION

```
cold:~/perl1$ ./sphere.pl
Volume of sphere of radius 7.5 = 1767.145865625
cold:~/perl1$
```

We couldn't just type in **pi**, because Perl doesn't know what it means, so we typed 3.14159265 instead. And rather than take the time to look up the precedence of the *exponentiation* operator (**), we put parentheses in to make it clear that we are cubing 7.5 and nothing else, even if precedence means the parentheses may not be necessary.

Scalar Variables

You know, if this was all we could do with Perl, we'd be better off buying a cheap calculator. But there's more to Perl than math operations. This is where *variables* come in: we can put numbers and other things into little chunks of memory and give them names. Perl has several kinds of variables; the most important and basic kind is the **scalar variable**. The word "scalar" means "has only one thing." A scalar variable stores exactly one thing: a number or a string.

Most languages don't make their variables look special in any way, but Perl uses punctuation characters as prefixes to make variables stand out. The punctuation character for scalars is the dollar sign, **$** (because it looks like an "S," for "scalar"). Following the **$** is a name that you (the programmer) select. Pick any name you like, as long as it follows these rules:

- It contains only uppercase letters, lowercase letters, digits, or underscores.
- It starts with a letter or an underscore.
- It can't be just a single underscore.

When we want to use a scalar variable in our program, we have to inform Perl of our plan by declaring the variable before we use it. We do that using the keyword **my**. We can also assign a value to the variable at the same time with the **=** operator.

Let's try an example. Create a new program named **sphere_variable.pl** file as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $radius;
$radius = 7.5;
my $PI = 3.14159265;
my $volume = 4/3 * $PI * ($radius ** 3);
print "Volume of sphere of radius ", $radius, " = ", $volume, "\n";
```

Check Syntax and run it:

INTERACTIVE TERMINAL SESSION

```
cold:~/perl1$ ./sphere_variable.pl
Volume of sphere of radius 7.5 = 1767.145865625
cold:~/perl1$
```

Did you get the same result as before? Hopefully you did.

Now, behold the power of variables!: Change the 7.5 to 49.4 and run the program again. Perl adjusts the output accordingly. We can also use scalars as items in the list of arguments we pass to **print()**.

In our programs, we'll assign values to variables as we declare them, like we did above with my $PI = 3.14159265;, but in mature programs, it's common to declare a scalar without giving it a value initially.

Only enter my *once per variable*. If you do it more than once, you'll get an error. Try it!

Let's try another example. Create the file **metric.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

# Convert meters to feet
my $feet = 5280;   # One mile
my $meters = $feet / 3.2808399;
print $feet, " feet is ", $meters, " meters\n";
```

Check Syntax and run it:

INTERACTIVE TERMINAL SESSION

```
cold:~/perl1$ ./metric.pl
5280 feet is 1609.3439975538 meters
cold:~/perl1$
```

Here you can see Perl's *comment* feature. Everything from a pound (#) sign on to the end of the line is a comment (this is one case where line breaks *are* significant to Perl). But if the pound (**#**) sign is inside of a string (between quotation marks), it's not a comment, it's just part of the string. Also, the shebang line at the beginning does not indicate a comment; it follows special rules that apply only when the first two characters in the file are #!.

Naming Conventions

In programming as in life, there are *rules* and then there are *conventions*. A rule *must* be followed or your program won't work. Following convention is not *required*, but they are recommended standards that programmers have agreed to adopt to make life easier for all.

So in addition to the rules to be adhered to when naming scalars, there are some conventions about naming them as well:

- Constants are typed in **$ALL_CAPITAL_LETTERS**.
- Local variables are **$lower_case**.
- Global variables are in **$Title_Case**.
- Words in the name are separated by underscores, as you see above.

These conventions differ from those recommended for some other popular programming languages, and while you could use those conventions in Perl, your programs would look like they were written with a funny accent.

We haven't yet covered the difference between local and global variables, so for the moment pretend they're all local variables. As for constants, we saw an example of one above: **$PI**.

Picking good names for variables is one of the most important skills a programmer has, so put some thought into it whenever you write a program.

Variables don't *have* to be initialized (given an initial value) when you declare them, mind you. Here's another program for you to write that demonstrates this. Create **from_metric.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

# Convert liters to various US measurements of volume
my $liters = 2.5;

my $LITERS_TO_FL_OZ   = 1000 / 29.574;
my $LITERS_TO_PINTS   = 1 / .4731;
my $LITERS_TO_GALLONS = $LITERS_TO_PINTS / 8;
my $LITERS_TO_CU_IN   = 1000 / 16.387;

print $liters, " liters = ",
      $liters * $LITERS_TO_FL_OZ,   " fluid ounces, ",
      $liters * $LITERS_TO_PINTS,   " pints, ",
      $liters * $LITERS_TO_GALLONS, " gallons, ",
      $liters * $LITERS_TO_CU_IN,   " cubic inches\n";
```

Check Syntax and run it:

INTERACTIVE TERMINAL SESSION

```
cold:~/perl1$ ./from_metric.pl
2.5 liters = 84.5337120443633 fluid ounces, 5.28429507503699 pints, 0.6605368843
79624 gallons, 152.559956062733 cubic inches
cold:~/perl1$
```

See how we split a very long **print()** statement over multiple lines there? We could have used multiple **print()** statements instead, like we did before, but as it says at the top of the famous camel book for Perl, "There's More Than One Way To Do It."

Take another look at our example code to see how we used spaces to make things line up vertically. Perl doesn't mind, and it makes the program easier to read.

Now we're going to break the program intentionally. Make the changes as shown in blue:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

# Convert liters to various US measurements of volume
my $liters;

my $LITERS_TO_FL_OZ   = 1000 / 29.574;
my $LITERS_TO_PINTS   = 1 / .4731;
my $LITERS_TO_GALLONS = $LITERS_TO_PINTS / 8;
my $LITERS_TO_CU_IN   = 1000 / 16.387;

print $liters, " liters = ",
      $liters * $LITERS_TO_FL_OZ,   " fluid ounces, ",
      $liters * $LITERS_TO_PINTS,   " pints, ",
      $liters * $LITERS_TO_GALLONS, " gallons, ",
      $liters * $LITERS_TO_CU_IN,   " cubic inches\n";
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./from_metric.pl
Use of uninitialized value in multiplication (*) at perl/from_metric.pl line 13.
Use of uninitialized value in multiplication (*) at perl/from_metric.pl line 13.
Use of uninitialized value in multiplication (*) at perl/from_metric.pl line 13.
Use of uninitialized value in multiplication (*) at perl/from_metric.pl line 13.
Use of uninitialized value in print at perl/from_metric.pl line 13.
 liters = 0 fluid ounces, 0 pints, 0 gallons, 0 cubic inches
```

It's a good thing we included **use warnings** in our code, because this program produces warnings! Perl complains about the use of an *uninitialized* value. In this case, we can see what's uninitialized: $liters. So, what value does it have? It has the special *undefined* value in Perl called **undef**. You can actually assign **undef** to a scalar explicitly.

But wait—Perl still executed the **print** statement, it just put zero in for every calculation. What's that all about? Well, the messages you saw were *warnings*; in other words, not a big enough a problem for Perl to stop running. (You might disagree, but Perl is a pretty promiscuous language!) When Perl sees **undef** used in a calculation, it produces a warning (as long as you have **use warnings** enabled) and then assumes a value of zero instead of **undef**, then just keeps going.

Finally, take the **use warnings** line out of the program and rerun it. The results look pretty bad, right? That's why we always put **use warnings** in every program.

You may have noticed that we haven't distinguished integers from floating point values in our programs. That's because Perl handles everything for us behind the scenes. If it can store a number in an integer, it does. If it needs a floating point value instead, it uses that. When an integer value needs to change to a floating point value (perhaps because it's being divided by a number that doesn't go into it evenly), Perl "promotes" it to a floating point number. Of course, some numbers can't be represented exactly in a computer because it only has a limited number of bits available for storing numbers (see the discussion about "precision" above).

You're doing great so far, and starting to get a real feel for the power of Perl. More cool stuff awaits you in the next lesson!

Conditional Statements in Perl

Lesson Objectives

When you complete this lesson, you will be able to:

- use conditional statements in Perl.
- use the operator **&&**.
- write tests in the proper order.
- use Perl's rules to determine whether an expression is true or false.

Conditional Statements in Perl

Life is full of uncertainty. If it's sunny, we'll go to the beach. If it rains, we'll go to the movies. If it snows, we'll go skiing.

In other words, we're going to change our plans depending on the conditions.

Perl's if Statement

And so it is with computer programs. Suppose we know someone's Body Mass Index and we want to make a value judgement about what that means, according to standardized rules:

BMI Range	Meaning
< 18.5	Underweight
18.5 - 25	Normal
25 - 30	Overweight
> 30	Obese

We can use Perl's **if** statement to accomplish the job. Let's explore. Create a program called **bmi.pl** as shown below:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $b_m_i = 23.4;

if ( $b_m_i < 18.5 )
{
  print "Underweight\n";
}
elsif ( $b_m_i < 25 )
{
  print "Normal\n";
}
elsif ( $b_m_i < 30 )
{
  print "Overweight\n";
}
else
{
  print "Obese\n";
}
```

Check Syntax and run it:

INTERACTIVE TERMINAL SESSION:

```
cold:~$ cd perl1
cold:~/perl1$ ./bmi.pl
Normal
cold:~$
```

Now, change the program as shown in blue:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

my $b_m_i = 14.1;

if ( $b_m_i < 18.5 )
{
  print "Underweight\n";
}
elsif ( $b_m_i < 25 )
{
  print "Normal\n";
}
elsif ( $b_m_i < 30 )
{
  print "Overweight\n";
}
else
{
  print "Obese\n";
}
```

Check Syntax and run it:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./bmi.pl
Underweight
cold:~/perl1$
```

The general form of the **if** statement is:

OBSERVE:

```
if ( condition )
{
   code...
}

...optionally followed by as many of these blocks as you want:

elsif ( condition )
{
   code...
}

...optionally followed by one of these blocks:

else
{
   code...
}
```

Those parentheses *must* be there; the curly braces *must* be there. (Later, you'll find out how to write the same kind of thing much more succinctly.)

Now let's change the order of the blocks in the **if** statement. Edit bmi.pl, adding the code shown in blue and deleting the code shown in ~~red~~:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

my $b_m_i = 23.4;

if ( $b_m_i < 18.5 )
{
  print "Underweight\n";
}
elsif ( $b_m_i < 30 )
{
  print "Overweight\n";
}
elsif ( $b_m_i < 25 )
{
  print "Normal\n";
}
~~elsif ( $b_m_i < 30 )~~
~~{~~
~~  print "Overweight\n";~~
~~}~~
else
{
  print "Obese\n";
}
```

Check Syntax and run it:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./bmi.pl
Overweight
cold:~/perl1$
```

See the difference?

Order matters because we took a shortcut in writing the conditions. $b_m_i is 23.4, so the first test (if ($b_m_i < 18.5)) will fail, but the next test (elsif ($b_m_i < 30)) will pass, and we'll run the code between the {} and print "Overweight\n." As we found the first true value, Perl then will consider itself done with the if block and move on to the next bit of code. This is a probem, because the <25 case will never be tested, and a person with a normal BMI will be mistakenly reported as Overweight! How do we make sure this doesn't happen?

Now suppose we want our tests to look more like the ones in the table from Wikipedia. We can do that using the operator **&&** ("and"—both sides have to be true). Add the blue code as shown:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

my $b_m_i = 23.4;

if ( $b_m_i < 18.5 )
{
  print "Underweight\n";
}
elsif ( $b_m_i >= 18.5 && $b_m_i < 25 )
{
  print "Normal\n";
}
elsif ( $b_m_i >= 25 && $b_m_i < 30 )
{
  print "Overweight\n";
}
elsif ( $b_m_i >= 30 )
{
  print "Obese\n";
}
else
{
  print "This shouldn't happen!!!\n";
}
```

Check Syntax and run it:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./bmi.pl
Normal
cold:~/perl1$
```

The four tests are complete; that is, there is no value that **$b_m_i** could take that wouldn't match one of them. But it's good practice when writing multi-way **if ... elsif ... elsif** constructions, to include an **else** clause, just in case you make an error in your logic. One common error that's made when writing the tests this way is to write **>** instead of **>=**. If **$b_m_i** happened to be *exactly* 25, we'd have a problem. That's why we usually write tests the way we did in the earlier example; it's easier to get the tests in the right order than to remember to put the equals signs in the right places.

The operators we've been using here for making conditions—**<**, **>**, **>=**, **&&**—are documented in **perldoc perlop** and http://perldoc.perl.org/perlop.html (look for "Relational Operators").

The **&&** operator deserves special mention—that's how we say "and" in a Perl condition, as in "this must be true *and* that must be true." To say "this must be true *or* this must be true, we use the "or" operator: **||**.

We've learned how to determine whether one number is greater or less than another. Now suppose we want to determine whether they're exactly equal? We do that using the **==** operator. Don't confuse this with the *assignment* operator we've already seen, **=**. They have to be different in Perl, otherwise if you wrote something like this code:

OBSERVE:

```
if ( $days = 7 )
```

That code would assign **7** to **$days** and use that as the value of the condition. While that action is actually legal in Perl, so many people were doing it by mistake that Perl now gives a warning if you put **=** inside of an **if** condition (yet another reason we include **use warnings** in all our programs).

Here's an example of the **==** and **||** operators in action:

OBSERVE:

```
if ( $b_m_i < 18.5 || $b_m_i > 25 )
{
  print "Abnormal\n";
}
elsif ( $b_m_i == 18.5 || $b_m_i == 25 )
{
  print "Right on the edge!\n";
}
else
{
  print "Normal\n";
}
```

This line would be read as: "Else if $b_m_i **is exactly equal to** 18.5 **or** $b_m_i **is exactly equal to** 25"...

What is Truth?

So now you've seen that **if** statements in Perl have *conditions* enclosed within parentheses, where we can ask a true or false question. In many languages, that's all you can use in their equivalents of the **if** statement. In Perl, you can enter any *expression*, that is, anything that could have a value. For instance, **$PI * ($radius**2)** is an expression. (Do you think it is true or false? Why?) You could place it within the parentheses of an **if** statement. Perl has rules to determine if an expression is true or false. Here they are:

- If a string expression evaluates to the empty string "" or the string "0" (the character zero and nothing else), then it's false.
- If a numeric expression evaluates to an integer zero or to a floating point zero (exactly zero, not "nearly" zero), then it's false.
- If an expression evaluates as the *undefined* value (remember that? "undef"), then it's false.
- Any other value of any kind of expression is true.

There you have it. The rule that the undefined value is false is pretty useful, because you'll recall that an uninitialized variable has the undefined value. That means you can use an **if** statement to figure out whether you've gotten a value for a variable yet. For instance:

OBSERVE:

```
my $year;

# Here's where the code that asks the user to enter a value for $year would go--
we'll see how
# to do that later--and if they don't enter a value, $year won't get changed.

unless ( $year )
{
  print "You didn't enter a year!\n";
}
```

If $year never got changed, it would still contain **undef** and therefore be false. (Of course, it would also be false if the user entered zero, but that's probably not useful either, since there wasn't a year zero, but I digress.)

See that **unless** keyword? That's Perl's handy way of saying "if the condition is *not* true." Don't try using **elsif** and **else** blocks with **unless**; Perl will execute those commands, but anyone reading your program—including you—might become horribly confused.

If you actually wanted to say "if the condition is *not* true," then use Perl's **not** (**!**) operator like this:

OBSERVE:

```
if ( ! $year )
{
  print "You didn't enter a year!\n";
}
```

Finally, suppose you want to be able to tell whether a variable hasn't yet been initialized or for some other reason contains the undefined value. There's a special function for that called **defined** that returns true only if its argument is defined, even if it's false:

OBSERVE:

```
my $x;
unless ( defined $x ) { print "Not defined!\n" }
$x = 0;
if ( defined $x )     { print "Defined!\n" }
```

Don't be afraid! Try these operators out and experiment a lot! Keep up the good work and see you in the next lesson!

Interpolation

Lesson Objectives

When you complete this lesson, you will be able to:

- assign strings to variables.
- interpolate values in strings.

Interpolation in Perl

Double-Quoted Strings

We refer to chunks of text as *strings* because they consist of characters *strung* together. We've been using strings to print text. We can also assign them to variables, like this:

```
my $title = "Teacher";
```

A *double-quoted string* wraps its contents between—believe it or not—double quotation marks. You can put anything you want between them, and your code will mean exactly what it says, with a few exceptions, like:

- The sequence **\n** means "line break here";
- The sequence **\t** means "move to next tab stop here";

Those two-character sequences—*digraphs* if you prefer the technical term—are called *escape sequences* in Perl (because the backslash character changes the meaning of the next character, thereby "escaping" it). If you want to include the actual backslash character itself, use *two backslashes:* \\.

Perl doesn't move the cursor to the next line or tab stop. Instead, Perl turns the escape sequences into *control characters*—special characters that are invisible—and stores those in the string. When something prints the string to a screen, the driver that decides where to put characters on the screen recognizes those control characters and interprets them in terms of cursor movement. According to convention, there are notional "tab stops" spaced every eight characters along a line; that convention is coded within the driver for the output device itself. So if you're printing a string on a printer, the escape sequence **\f** gets turned by Perl into the control character called form-feed, and the printer will produce a new page.

Some escape sequences don't get turned into control characters, but instead cause Perl to change what follows them:

- **\l**: lowercase next character
- **\L**: lowercase all characters until end of string or the sequence **\E**
- **\u**: uppercase next character
- **\U**: uppercase all characters until end of string or the sequence **\E**

Hmm. That seems a little weird; after all, if you want the next character to be uppercase, why not type it that way? We'll learn the answer to that question shortly.

There are a few other escape sequences of diminishing usefulness; see **perldoc perlop** or http://perldoc.perl.org/perlop.html, "Quote and Quote-like Operators."

But double-quoted strings are capable of much more. You can put a scalar variable inside one, and the value of the variable will be substituted—in Perl we say *interpolated*—at that point.

Let's see what that looks like. Create a new file named **insect_sales.pl** and type the code below as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $date        = localtime();
my $ants        = 47_000;
my $fleas       = 240_000;
my $beetles     = 520;
my $fruit_flies = 1_500_000;
print "Welcome to Echidna Eric's Insect Emporium\n";
print "\n";
print "This is the inventory stock report for $date\n";
print "---------------------------------------------------------\n";
print "We have $ants ants\n";
print "We have $fleas fleas\n";
print "We have $beetles beetles\n";
print "We have $fruit_flies fruit flies\n";
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~$ cd perl1
cold:~$ ./insect_sales.pl
Welcome to Echidna Eric's Insect Emporium

This is the inventory stock report for Thu Dec  3 14:55:14 2009
---------------------------------------------------------
We have 47000 ants
We have 240000 fleas
We have 520 beetles
We have 1500000 fruit flies
cold:~/perl1$
```

Perl knows the current date and time! That's a feature of the **localtime()** function—assign its result to a scalar and you get that handy string.

We added underscores to large numbers to make them more readable—the underscores are where commas (or in Europe, the decimal points) would be if I was typing the numbers for publication, like **12,345**. We can't use commas or periods here, because they mean something else in Perl, so we use underscores instead.

Every variable inside a double-quoted string is replaced by its current value at run-time. This *interpolation* makes **print** statements more readable. Compare these two statements:

print "Name:\t", $name, ", phone:\t", $phone, "\n";

or

print "Name:\t$name, phone:\t$phone\n";

A few things about the sample code above might bother you. If you typed the code in by hand, you probably got tired of all those **print** statements. Any time you spot repetition in a program, that's a sign that there's probably a better way to do it. How could we collapse all of those **print** statements down to one?

We could run all the strings together, but the program would become far less *readable*:

print "Welcome to Echidna Eric's Insect Emporium\n\nThis is the inventory stock report for $date\n";

This is where Perl's *here documents* come in handy. (We call them *heredocs* for short.) What we'd really like to write in the code is the text to be printed out, the way we want it to come out; heredocs let us do this. Edit the program as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $date        = localtime();
my $ants        = 47_000;
my $fleas       = 240_000;
my $beetles     = 520;
my $fruit_flies = 1_500_000;
print <<"END_OF_REPORT";
Welcome to Echidna Eric's Insect Emporium

This is the inventory stock report for $date
----------------------------------------------------------
We have $ants ants
We have $fleas fleas
We have $beetles beetles
We have $fruit_flies fruit flies
END_OF_REPORT
print 'done';
```

Check Syntax and run it.

Running the program gives the same result, but it's much more readable now. Here's how it works: **<<**, followed by a string within double quotation marks (in this case, **END_OF_REPORT**), is a Perl expression. That expression's value is the string that starts on the next line and continues up until (but not including) a line consisting of *precisely* the same string that followed **<<**. You must then have a newline character and nothing else. That includes trailing spaces, so make sure you don't have any! If you do, or if you make some other mistake in the terminating line, Perl will run off the end of your program searching for it and then complain. Let's make that mistake deliberately so you know what the error looks like. Edit your code as shown:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

my $date        = localtime();
my $ants        = 47_000;
my $fleas       = 240_000;
my $beetles     = 520;
my $fruit_flies = 1_500_000;
print <<"END_OF_REPORT";
Welcome to Echidna Eric's Insect Emporium

This is the inventory stock report for $date
----------------------------------------------------------
We have $ants ants
We have $fleas fleas
We have $beetles beetles
We have $fruit_flies fruit flies
END_OF_THE_REPORT
print 'done';
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./insect_sales.pl
Can't find string terminator "END_OF_REPORT" anywhere before EOF at ./insect.pl
line 10.
cold:~/perl1$
```

Now we've got a problem. Let's change it back so it runs correctly. Remove the ~~red~~ text as shown:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

my $date        = localtime();
my $ants        = 47_000;
my $fleas       = 240_000;
my $beetles     = 520;
my $fruit_flies = 1_500_000;
print <<"END_OF_REPORT";
Welcome to Echidna Eric's Insect Emporium

This is the inventory stock report for $date
---------------------------------------------------------
We have $ants ants
We have $fleas fleas
We have $beetles beetles
We have $fruit_flies fruit flies
END_OF_~~THE_~~REPORT
print 'done';
```

The string behaves like a *double-quoted* string, so you can interpolate variables in it.

Note You could embed literal newlines within a double-quoted string and print the whole report with one **print** statement and one set of double quotes around everything. But if you include a double quote somewhere in the report without realizing it, the quote would have to be escaped. It's also harder to see where a multi-line string ends without a distinctive line, like the one heredocs supplies.

Edit the program as shown:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

my $date        = localtime();
my $ants        = 47_000;
my $fleas       = 240_000;
my $beetles     = 520;
my $fruit_flies = 1_500_000;

print <<"END_OF_REPORT";
Welcome to Echidna Eric's Insect Emporium

This is the inventory stock report for $date
---------------------------------------------------------
We have $ants ants
We have $fleas fleas
We have $beetles beetles
We have $fruit_flies fruit flies
END_OF_REPORT
my $insect_of_the_month = "caterpillar";
 my $insect_of_the_month_count = 1_200;
 print "This month, we have $insect_of_the_month_count $insect_of_the_months\n";
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./insect_sales.pl
Global symbol "$insect_of_the_months" requires explicit package name at ./insect
.pl line 23.
Execution of ./insect.pl aborted due to compilation errors.
cold:~/perl1$
```

Hmmm. It doesn't work. There is no variable named **$insect_of_the_months**. We meant to print out the value of **$insect_of_the_month** followed by an 's'. But Perl doesn't know that-it just reads as many consecutive characters in a variable name as it can, while still following the rules, and used that result. To get the result we want, we need some syntax in Perl that indicates, "this is the end of a variable name." For that, we'll use braces {}. Edit **insect_sales.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $date        = localtime();
my $ants        = 47_000;
my $fleas       = 240_000;
my $beetles     = 520;
my $fruit_flies = 1_500_000;

print <<"END_OF_REPORT";
Welcome to Echidna Eric's Insect Emporium

This is the inventory stock report for $date
---------------------------------------------------------
We have $ants ants
We have $fleas fleas
We have $beetles beetles
We have $fruit_flies fruit flies
END_OF_REPORT
my $insect_of_the_month = "caterpillar";
 my $insect_of_the_month_count = 1_200;
 print "This month, we have $insect_of_the_month_count ${insect_of_the_month}s\n
";
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./insect_sales.pl
This month, we have 1200 caterpillars
cold:~/perl1$
```

So now you've seen how variables can be interpolated inside double-quoted strings; how do you think the **\l** and **\U** escape sequences are useful? Think it over before reading the next section.

Single-Quoted Strings

> Quotation, n.: The act of repeating erroneously the words of another. The words erroneously repeated. - Ambrose Bierce, The Devil's Dictionary

Let's say you want an actual, uninterpolated, plain dollar sign **$** in a double-quoted string. You'd escape it in the usual way: **"I have \$10 in my wallet"**. And suppose you want a lot of plain dollar signs and backslash characters that remain as such. You could accomplish this by escaping each one, but you're going to get tired of hitting the backslash key, not to mention reading that long string!

You might think that this type of situation doesn't come up very often, outside of financial reports, but there's one example of strings like that you're certain to encounter: Perl code itself. We're not ready to tackle Perl code embedded as strings in Perl programs just yet, but trust me, they do occur, and when they do, you'll be glad you learned about *single-quoted* strings.

A single-quoted string is surrounded by single quotation marks and obeys these basic rules:

- A backslash followed by single quote (**\'**) means single quote (**'**)
- A backslash followed by backslash (\\) means backslash (****)

That's it. Everything else comes out just the way you put it in—including any backslashes that aren't followed by a single quotation mark or a backslash. Heredocs follow the same rules—use a single quotation mark after the **<<** instead of double quotation marks. In that case, *everything* inside the heredoc is taken literally, including all backslashes and single quotation marks. This can be especially useful when you want to print a chunk of text completely literally—with no interpolation of any kind—and you don't want to escape dollar signs or check for other things that can be interpolated.

Take a look at this code:

OBSERVE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $mail_message = <<'END_OF_TEXT';
From: Dutiful Developer <programmer@example.com>
To: Masterful Manager <boss@example.com>
Subject: Code finished

Hi boss, I finished my program.  I'm especially proud of this part:

  my $first_name = "creative";
  my $last_name = "Ceo";
  print "This program ($0) is dedicated to our dear leader, \L\u$first_name \U$l
ast_name\n";
END_OF_TEXT
```

There's some interesting stuff embedded in the code written by Dutiful Developer that I haven't told you about yet. Take the code fragment that's quoted in the mail message in the program (we have a program within a string within a program...), put it in a new file called **escape_test.pl**, and *add the necessary preamble for all programs*, as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $first_name = "creative";
my $last_name = "Ceo";
print "This program ($0) is dedicated to our dear leader, \L\u$first_name \U$las
t_name\n";
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./escape_test.pl
This program (./escape_test.pl) is dedicated to our dear leader, Creative CEO

cold:~/perl1$
```

There's a detailed explanation of what's happening in this code here: http://perldoc.perl.org/perlvar.html; look for "PROGRAM_NAME." Take note of the effect of **\L\u**. Do you see why Perl's creator would want it to work this way?

Also, our code has one of Perl's *special variables*—**$0**. It's not a variable *you* can declare—it doesn't comply with the naming rules we know. But it's one of a number of variables that Perl populates with certain values. **$0** contains the name of the program being executed (including whatever path was specified in the command line invocation).

Interpolate Now Or Later?

"Believe me, you have to get up early if you want to get out of bed"
-Groucho Marx

Modify **escape_test.pl** as follows:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

my $first_name = "creative";
my $last_name = "Ceo";

my $dedication = "This program ($0) is dedicated to our dear leader, \L\u$first_
name \U$last_name\n";
$first_name = 'passionate';
$last_name = 'president';
print $dedication;
```

Check Syntax and save it—but *before you run it*—quick, what do you think it will do? Will the dedication be to "Creative CEO" or "Passionate President"? *Why* do you think that?

Now run the program. You can see from the result that Perl does what we call *early binding* of variable values. That is, when **$dedication** is assigned, the variables that are part of the expression assigned to it are evaluated to their values *in that instant*. Changing those variables later makes *no difference* to the value of **$dedication**, because it's already been formed.

(And thank goodness it works this way, too; otherwise Perl programming would become seriously difficult. Think about it.)

Wow. We're really making progress. Keep it up! See you in the next lesson!

Strings and String Conditionals

Lesson Objectives

When you complete this lesson, you will be able to:

- use strings to manipulate text.
- use separate operators to compare strings.
- translate numbers into strings and strings into numbers.

String Conditions and String Functions

Welcome back! In this lesson, you'll learn about using strings in Perl to manipulate text.

String Conditions

We've already compared numeric quantities in Perl. Here's a quick refresher:

Operator	Example	Meaning
==	$x == $y	$x equals $y
!=	$x != $y	$x does not equal $y
<	$x < $y	$x is less than $y
>	$x > $y	$x is greater than $y
<=	$x <= $y	$x is less than or equal to $y
>=	$x >= $y	$x is greater than or equal to $y

There are equivalent operators for testing strings:

Operator	Example	Meaning
eq	$x eq $y	$x equals $y
ne	$x ne $y	$x does not equal $y

Some additional numeric equivalent operators in Perl are: **lt**, **gt**, **le**, and **ge**. I didn't include them in the table above though, because you will almost never use them—in nearly 20 years of Perl programming I have used each of them maybe once. (We use different operators for sorting, but we'll get to them later.)

The other Perl operators you've met so far are composed of punctuation characters or symbols like **<**. These (and other) operators are composed of letters and so you should leave white space around them. This makes it easier for people to read and avoids the possibility of syntax errors, such as you'd get if you wrote this code: **xeqy**. Perl would look for a variable **$xeq**. (Perl reads to find the next complete variable name, even if that variable doesn't exist), and that variable would then be followed by another variable **$y**, which cannot follow a variable in Perl's *grammar* (the language that describes the rules that apply to Perl programs).

Why do we need separate operators to compare strings in Perl? Since a scalar can contain either a string or a number, and Perl knows which type of item is stored in the scalar, wouldn't one set of operators do? Then we could just let Perl figure out whether to compare things numerically or lexicographically, right?

These are all really good questions—the kind that keep language designers awake nights. Here's why we need two sets of operators. Suppose **$x** was a string and **$y** was a number, and we were using a generic operator to compare them. Would Perl compare them numerically, or lexicographically?

Perl helps by "coercing" (translating) numbers into strings and vice-versa. If a number is stored in a scalar and you're applying a string operator or function to it, Perl will turn the number into a string first. That string will yield the same result as if you printed the scalar (because **print** is a string function). Numbers get printed—and stringified—in a specific format (truncated after a certain number of digits, trailing zeroes after a decimal point suppressed, and such). And if a string is stored in a scalar and you use a numeric operator or function on it, Perl turns it into a number first. Now every number can be turned into a string. But not every string can be turned into a number. The string **"123"** can be turned into the number **123**. The string **"-42.87248233449873687634904095912249135"** can be turned into a number, but we lose the digits that

Perl doesn't have the precision to store.

But what should Perl do with the string **"petunia"** when asked to turn it into a number? Perl has some helpful rules in this kind of situation. It reads a string and as long as it finds characters that are legally part of a number, it keeps going. If it has to give up before the end of the string, it returns whatever it found up to that point, and if **use warnings** is turned on, it gives a warning. So the string **"21 Jump Street"** evaluates as the number **21** with a warning. If Perl doesn't read *any* legal numeric characters, then it returns **undef** which, when treated as a number, *evaluates as zero*. So **"petunia" + 42** will evaluate to **42**, with a warning **argument "petunia" isn't numeric in addition** (so long as **use warnings** is on). It's important to turn on **use warnings**, otherwise you'd have no idea that you were trying to add a number to a flower!

So even though Perl lets us store a number in a scalar and later store a string in the same scalar, it's important that we know the type of item we expect to be stored in any variable when we prepare to perform a comparison on it. Perl's coercion rules help us read values from somewhere else, when they are always read as strings, without having to waste time converting elements that we know are numbers.

Let's do a straightforward string comparison example so you get the hang of it. Create the file called **fruit_check.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $fruit;
$fruit = 'apple';

if ( $fruit eq 'lemon' || $fruit eq 'orange' || $fruit eq 'lime' )
{
  print "Citrus... juicy!\n";
}
elsif ( $fruit eq 'strawberry' || $fruit eq 'raspberry' || $fruit eq 'blackberry' || $fruit eq 'loganberry' )
{
  print "Berry...\n";
  if ( $fruit ne 'loganberry' )
  {
    print "pie!\n";
  }
}
else
{
  print "Some other kind of fruit!\n";
}
```

and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~$ cd perl1
cold:~/perl1$ ./fruit_check.pl
Some other kind of fruit!
cold:~/perl1$
```

Experiment with the program by assigning the **$fruit** variable to a string other than **'apple'**. Use some of the other fruits mentioned in the program. We used the **||** operator to mean "or" in a logical expression, and the **eq** and **ne** operators to compare strings.

String Functions and Operators

In Perl, we can join strings together, make them uppercase and lowercase, find out how long they are—the list goes on...

The function **lc()** makes strings lowercase. The function **uc()** makes strings uppercase. These functions are really useful when you want to compare strings without regard to case. Modify **fruit_check.pl** as shown:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

my $fruit;
$fruit = 'Orange';

if ( $fruit eq 'lemon' || $fruit eq 'orange' || $fruit eq 'lime' )
{
  print "Citrus... juicy!\n";
}
elsif ( $fruit eq 'strawberry' || $fruit eq 'raspberry'
     || $fruit eq 'blackberry' || $fruit eq 'loganberry' )
{
  print "Berry...\n";
  if ( $fruit ne 'loganberry' )
  {
    print "pie!\n";
  }
}
else
{
  print "Some other kind of fruit!\n";
}
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./fruit_check.pl
Some other kind of fruit!
cold:~/perl1$
```

Oops! **'Orange'** is the same as **'orange'** for *our* purposes, but if we want a computer to think they're equivalent, we have to explain it. Modify the program as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $fruit;
$fruit = 'Orange';

my $fruit_lc = lc( $fruit );
if ( $fruit_lc eq 'lemon' || $fruit_lc eq 'orange' || $fruit_lc eq 'lime' )
{
  print "Citrus... juicy!\n";
}
elsif ( $fruit_lc eq 'strawberry' || $fruit_lc eq 'raspberry'
     || $fruit_lc eq 'blackberry' || $fruit_lc eq 'loganberry' )
{
  print "Berry...\n";
  if ( $fruit_lc ne 'loganberry' )
  {
    print "pie!\n";
  }
}
else
{
  print "Some other kind of fruit!\n";
}
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./fruit_check.pl
Citrus... juicy!
cold:~/perl1$
```

Do you see how that works? By creating **$fruit_lc** as a lowercase version of **$fruit**, we can compare it to any lowercase string, such as **'orange'**. (It would have worked equally well to use **uc()** and compare to **'ORANGE'**; just make sure you're consistent!)

We didn't use **$fruit = lc($fruit)** just in case we want to use the *original* version of **$fruit** to do something else later, like print out in an error message.

Now let's see how to *concatenate* strings. Concatenation is a technical term that means "join together." You already know one way to join two strings—say **$x** and **$y**—together: interpolation (**"xy"**). So why do we have another way of doing it? Well, Perl is all about choice—remember the Perl maxim, "There's more than one way to do it." Sometimes, concatenation makes code easier to read and/or more logical than interpolation. Here's how we'd concatenate **$x** and **$y**:

$x . $y

See that dot in between **$x** and **$y**? That's the concatenation operator. The result of the expression above is the two variables (stringified if one of them is a number) joined together.

Here's a great reason to use the concatenation operator. Say you're constructing a really long string to print as part of a report:

OBSERVE:

```
$title = "Observations on the Mating Habits of $bird with Special Reference to N
esting Behavior Documented in $country";
```

That's a very wide line, longer than most best practices of 72 characters, and far too long if you're using it in a magazine article, for instance. The concatenation operator comes to the rescue:

OBSERVE:

```
$title = "Observations on the Mating Habits of $bird"
        . " with Special Reference to Nesting Behavior"
        . " Documented in $country";
```

Now we can break the line up any way we please.

Okay, now suppose we're printing **$title** in a report and we need to know whether it will fit on the output line. How can we tell, when the value of **$bird** could be anything from "Auk" to "Himalayan Golden Backed Three-Toed Woodpecker"? This is where the **length** function comes in handy. It tells us how many characters are in a string:

length("Auk") == 3 # True!

WARNING

For the purposes of our discussions in this course, we're dealing only with characters in the ASCII set and therefore "character" is synonymous with "byte." That is not true when dealing with Unicode—an extended character set capable of expressing just about every character in just about every language—but that's outside the scope of this course.

Here's another way we might use the length function. It can check to see whether a variable looks like a US zip code. Create **zip_check.pl** and enter the code as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $zip;
$zip = "98362";

if ( length( $zip ) == 5 )
{
  print "$zip could be a standard zip code\n";
}
elsif ( length( $zip ) == 9 || length( $zip ) == 10 )   # May have hyphen between
 parts
{
  print "$zip could be a ZIP+4 code\n";
}
else
{
  die "$zip is not a zip code\n";
}
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./zip_check.pl
98362 could be a standard zip code
cold:~/perl1$
```

Try setting **$zip** to **"4077"**, **"12345-1234"**, and **"hedgehogs"** and rerun it after each change. You can see that the check isn't perfect, but it's a start.

We put the zip code in a string, now let's see what happens when you change it to a number, for instance **95472** (O'Reilly's west coast office zip code). Now try the number **04401** (the zip code for Bangor, Maine). Look closely at the output. What's odd about it? Try the number **02138** (O'Reilly's east coast office zip code). Remember—don't put quotes around a number! What happens?

You've just learned how to input numbers in a different *radix* or *base*—in this case, *octal* (base 8). For more information, check out **perldoc perldata** or http://perldoc.perl.org/perldata.html, particularly the section titled "Scalar value constructors." Try changing **$zip** to numbers like **0xface0ff** and **0b11001001**—again (remember, in Perl there are no quotation marks around numbers). See what gets printed and make sure you understand the reasons behind it. Then, put quotation marks around those values—**"0xface0ff"** and **"0b11001001"**—and see what happens. Change those values until you're confident that you understand the difference between a string and a number.

You Know Things About Strings

I'm feeling confident in your progress so far. Good job! It may feel like we're learning Perl slowly—and indeed we are taking baby steps—but by taking time to understand the fundamentals, we'll have a strong foundation to build upon. When people don't learn these fundamentals well, they end up writing only "cargo cult code"—code assembled by copying and pasting from other people's examples, then hoping for the best.

Look up "cargo cult code" in Wikipedia, to read more about the kind of programmer you will not be!

See you in the next lesson!

Lists and List Functions

Lesson Objectives

When you complete this lesson, you will be able to:

- differentiate between lists and scalars.
- use various list functions.

"Enough organization, enough lists, and I think we can control the uncontrollable."
-John Mankiewicz

Most people run their lives using lists, like shopping lists, calendars, address books, even lists of the lists they've made!

Lists

Perl has lists, too. You've been using them for a while. In this lesson, you're going to get to know and understand them a lot better.

There's an important distinction to make about lists: they are *not* variables like scalars are. Lists hold data, but they don't have *names*. You can store a list in a variable (not a scalar, but a different kind of variable. We'll see how to do that in the next lesson). But a list itself is an ordered set of scalar *values* flying around between different parts of your program.

A list shows up in your program in these ways:

- When you assign something to a series of scalars between parentheses separated by commas:

  ```
  ($first_name, $middle_initial, $last_name) = ...
  ```

  The set of scalars there is the list. Whatever is on the right side of the equals sign will be expected to produce a list that can be assigned to those scalars.

- When you pass arguments to a function that expects a list (most of them do), like **print**:

  ```
  print $first_name, " ", $middle_initial, ". ", $last_name;
  ```

  The parentheses are optional in this case.

There's more to the whole notion of functions *expecting* a list, but for now let's keep building on the stuff we know. (We have an entire lesson dedicated to the intricacies of lists later.)

We can *declare* more than one variable at a time in a list:

```
my ($x, $y, $z);
```

We can *assign* values to more than one variable at a time in a list:

```
($x, $y, $z) = (17, 24, 42);
```

We can *initialize* (declare and assign at the same time) in a list:

```
my ($x, $y, $z) = (17, 24, 42);
```

The contents to the right side of the equals sign in the last two examples are also lists. They contain *literal* elements—that's the technical term for the values specified at that point in the program, not coming from a variable.

Let's check out an example to see what happens when the number of things on the left side is different from the number of things right side of the equals sign. Create a new file called **list_test.pl**as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my ($a, $b, $c, $d);
($a, $b) = (17, 24, 42);  # Too few targets
print "\$a = $a, \$b = $b\n";
($a, $b, $c, $d) = (17, 24, 42);   # Too many targets
print "\$a = $a, \$b = $b, \$c = $c, \$d = $d\n";
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~$ cd perl1
cold:~/perl1$ ./list_test.pl
$a = 17, $b = 24
Use of uninitialized value $d in concatenation (.) or string at line 9.
$a = 17, $b = 24, $c = 42, $d =
cold:~/perl1$
```

What happened? Ponder and absorb.

split()

There are two functions that convert between a string and a list. Let's start with **split()**. Create a file called **store.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

# Grocery store inventory:
my $lines = <<'END_OF_REPORT';
0.95  300   Peaches
1.45  120   Avocados
5.50   10   Durien
0.40  700   Apples
END_OF_REPORT

my ($line1, $line2, $line3, $line4) = split "\n", $lines;
my ($cost, $quantity, $item) = split " ", $line1;
print "Total value of $item on hand = \$", $cost * $quantity, "\n";
($cost, $quantity, $item) = split " ", $line2;
print "Total value of $item on hand = \$", $cost * $quantity, "\n";
($cost, $quantity, $item) = split " ", $line3;
print "Total value of $item on hand = \$", $cost * $quantity, "\n";
($cost, $quantity, $item) = split " ", $line4;
print "Total value of $item on hand = \$", $cost * $quantity, "\n";
```

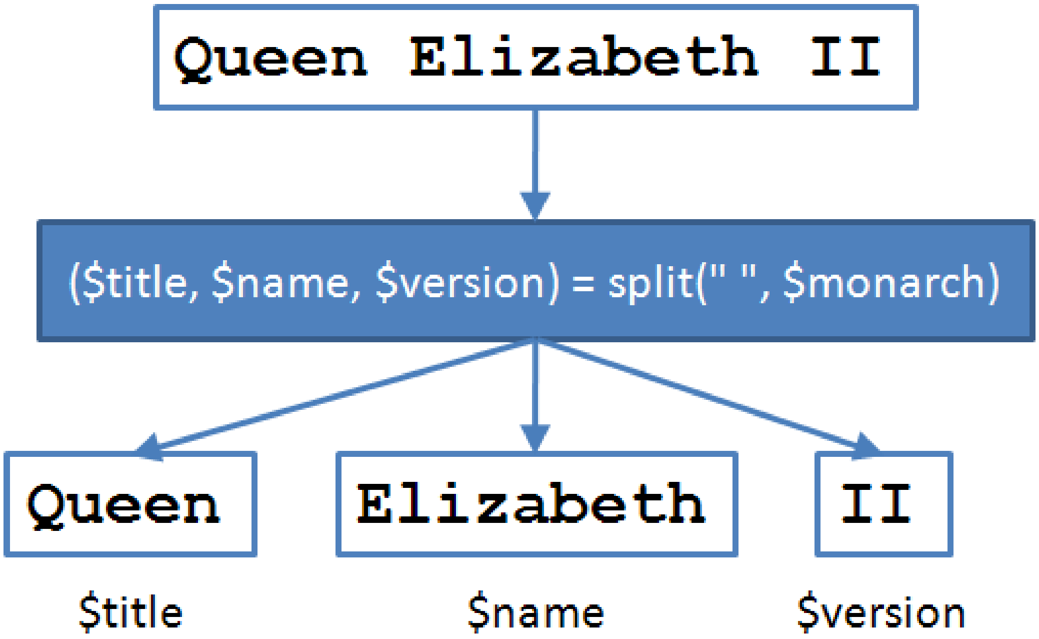
Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./store.pl
Total value of Peaches on hand = $285
Total value of Avocados on hand = $174
Total value of Durien on hand = $55
Total value of Apples on hand = $280
cold:~/perl1$
```

split() works by splitting the *second* argument wherever it sees an instance of the *first* argument, and returning the *list* of strings that it finds. (By the way, when I'm writing about a function, often I'll put parentheses after it like I just did, to help you realize that I'm talking about a function. It doesn't mean anything about whether or not parentheses are required - you'll see I used none in the code above.)

Here's a graphical representation:



The first argument—first a newline, then a space, in the above example—is *not* included in the resulting strings. Instead, **split()** returns the things in between. Also, **split()** does not modify the argument you pass it—even though Perl programmers may refer to those strings as having been being split, the original string has not actually changed.

Also, when using **split()**, the string you use to execute a split (the first argument) does not have to be a single character. If, however, that string is a single space (like we used in our example above), then Perl will split its second argument wherever it sees *any number* of consecutive spaces or tab characters as the string. That's the reason splitting the grocery store lines on a single space worked, even though there were multiple spaces between the items in the string.

Let's try one last example using **split()**; change **store.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

# Grocery store inventory:
my $lines = <<'END_OF_REPORT';
0.95  300   White Peaches
1.45  120   California Avocados
5.50   10   Durien
0.40  700   Spartan Apples
END_OF_REPORT

my ($line1, $line2, $line3, $line4) = split "\n", $lines;
my ($cost, $quantity, $item) = split " ", $line1;
print "Total value of $item on hand = \$", $cost * $quantity, "\n";
($cost, $quantity, $item) = split " ", $line2;
print "Total value of $item on hand = \$", $cost * $quantity, "\n";
($cost, $quantity, $item) = split " ", $line3;
print "Total value of $item on hand = \$", $cost * $quantity, "\n";
($cost, $quantity, $item) = split " ", $line4;
print "Total value of $item on hand = \$", $cost * $quantity, "\n";
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./store.pl
Total value of White on hand = $285
Total value of California on hand = $174
Total value of Durien on hand = $55
Total value of Spartan on hand = $280
cold:~/perl1$
```

This output isn't very useful, is it? Do you understand why you got it? (Hint: Take a look at the last example of lists you created.) So, how can we fix it? There's a third, optional argument for **split()** that tells Perl to stop splitting after it reaches a specified number of fields.

Modify **store.pl** again as shown:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

# Grocery store inventory:
my $lines = <<'END_OF_REPORT';
0.95  300   White Peaches
1.45  120   California Avocados
5.50   10   Durien
0.40  700   Spartan Apples
END_OF_REPORT

my ($line1, $line2, $line3, $line4) = split "\n", $lines;
my ($cost, $quantity, $item) = split " ", $line1, 3;
print "Total value of $item on hand = \$", $cost * $quantity, "\n";
($cost, $quantity, $item) = split " ", $line2, 3;
print "Total value of $item on hand = \$", $cost * $quantity, "\n";
($cost, $quantity, $item) = split " ", $line3, 3;
print "Total value of $item on hand = \$", $cost * $quantity, "\n";
($cost, $quantity, $item) = split " ", $line4, 3;
print "Total value of $item on hand = \$", $cost * $quantity, "\n";
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./store.pl
Total value of White Peaches on hand = $285
Total value of California Avocados on hand = $174
Total value of Durien on hand = $55
Total value of Spartan Apples on hand = $280
cold:~/perl1$
```

We cannot interpolate an *expression* inside a string and expect it to expand. Modify your code as shown:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

# Grocery store inventory:
my $lines = <<'END_OF_REPORT';
0.95  300   White Peaches
1.45  120   California Avocados
5.50   10   Durien
0.40  700   Spartan Apples
END_OF_REPORT

my ($line1, $line2, $line3, $line4) = split "\n", $lines;
my ($cost, $quantity, $item) = split " ", $line1, 3;
print "Total value of $item on hand = ~~\$", ~~$cost * $quantity~~, "~~\n";
($cost, $quantity, $item) = split " ", $line2, 3;
print "Total value of $item on hand = \$", $cost * $quantity, "\n";
($cost, $quantity, $item) = split " ", $line3, 3;
print "Total value of $item on hand = \$", $cost * $quantity, "\n";
($cost, $quantity, $item) = split " ", $line4, 3;
print "Total value of $item on hand = \$", $cost * $quantity, "\n";
```

Check Syntax and run it. See what happened?

Also note that once we have declared a variable, we do not declare it again. Let's find out what happens if we try adding **my** in front of any of the lines that assign to the list **($cost, $quantity, $item)**. Modify your code as shown:

CODE TO EDIT:

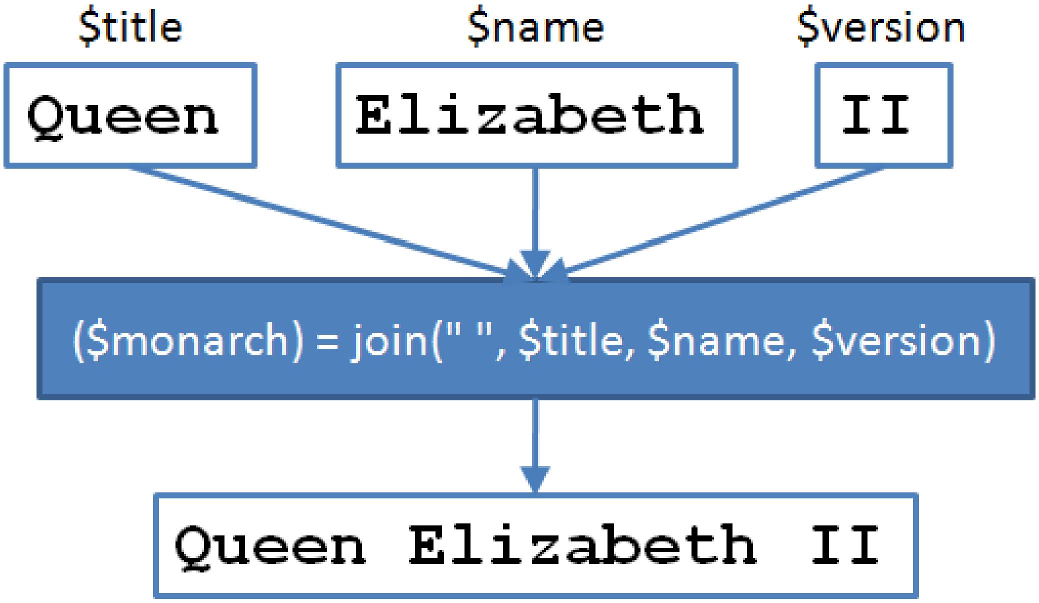
```
#!/usr/bin/perl
use strict;
use warnings;

# Grocery store inventory:
my $lines = <<'END_OF_REPORT';
0.95  300   White Peaches
1.45  120   California Avocados
5.50   10   Durien
0.40  700   Spartan Apples
END_OF_REPORT

my ($line1, $line2, $line3, $line4) = split "\n", $lines;
my ($cost, $quantity, $item) = split " ", $line1, 3;
print "Total value of $item on hand = \$", $cost * $quantity, "\n";
my ($cost, $quantity, $item) = split " ", $line2, 3;
print "Total value of $item on hand = \$", $cost * $quantity, "\n";
($cost, $quantity, $item) = split " ", $line3, 3;
print "Total value of $item on hand = \$", $cost * $quantity, "\n";
($cost, $quantity, $item) = split " ", $line4, 3;
print "Total value of $item on hand = \$", $cost * $quantity, "\n";
```

Check Syntax and run it. What error messages do you see? (By the way, if you're a tad annoyed by the duplication of that code doing the same thing four times, you should be! But let it go for now; we'll show how to deal with that later in the course.)

join()

join() is the *converse* of **split()**. The first argument we'll **join()** is a string that Perl will place in between each of the remaining arguments (if any of them are numbers, they'll be "stringified" first), returning the resulting string. The joining string is not placed before the first argument, or after the last argument, it is only in between arguments.

Here's a graphical representation:



Let's modify our grocery store example so that after it splits the lines, it subtracts some sales and then prints out the result using **join()**.

Create a file called **store_join.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $lines = <<'END_OF_REPORT';
0.95  300   White Peaches
1.45  120   California Avocados
5.50   10   Durien
0.40  700   Spartan Apples
END_OF_REPORT

my ($peaches_sold, $avocados_sold, $durien_sold, $apples_sold) = (12, 6, -1, 24);  # On
e durien returned... too smelly
my ($line1, $line2, $line3, $line4) = split "\n", $lines;
my ($cost, $quantity, $item) = split " ", $line1, 3;
$quantity -= $peaches_sold;
$line1 = join " ", $cost, $quantity, $item;
print "$line1\n";
($cost, $quantity, $item) = split " ", $line2, 3;
$quantity -= $avocados_sold;
$line2 = join " ", $cost, $quantity, $item;
print "$line2\n";($cost, $quantity, $item) = split " ", $line3, 3;
$quantity -= $durien_sold;
$line3 = join " ", $cost, $quantity, $item;
print "$line3\n";
($cost, $quantity, $item) = split " ", $line4, 3;
$quantity -= $apples_sold;
$line4 = join " ", $cost, $quantity, $item;
print "$line4\n";
```

and run it.

INTERACTIVE TERMINAL SESSION:

```

cold:~/perl1$ ./store_join.pl
0.95 288 White Peaches
1.45 114 California Avocados
5.50 11 Durien
0.40 676 Spartan Apples
cold:~/perl1$
```

The lines that are produced aren't quite the same as the ones that were entered. The numbers are different and there is only one space between each item, because that's what we told **join()** to do. Because that output can be parsed by the same program, the number of spaces usually doesn't matter. If, however, some other (less tolerant) program expects the lines to be composed of fixed-width fields, then we'd have a problem (one that we will learn to solve later).

Incidentally, a list can have *zero* or more things in it. Here's what a list with nothing in it looks like:

()

And a list with only one thing in it looks like this:

("thing")

Binary Operator Shortcut

One final bit of instruction for you in this lesson—you see that **-=** in the program? That's a handy Perl shortcut to use when you want to apply a *binary* operator (an operator that takes two arguments, one on each side—like **+**, or *****) and then have the variable on the left side of the operator be assigned the result of the calculation; in other words and code:

$variable *OP= EXPRESSION*

is the same as writing:

$variable = **$variable** *OP EXPRESSION*

where *OP* is any binary operator, and *EXPRESSION* is any expression. So:

$quantity -= **$peaches_sold**

is the same as:

$quantity = **$quantity** - **$peaches_sold**

This shortcut saves us time when programming and makes our code very readable.

Looking great so far! In the next lesson we'll learn about arrays! See you there!

Arrays and Array Functions

Lesson Objectives

When you complete this lesson, you will be able to:

- declare arrays.
- assign array elements.
- assign an array to a list.
- copy one array to another.
- use array functions.

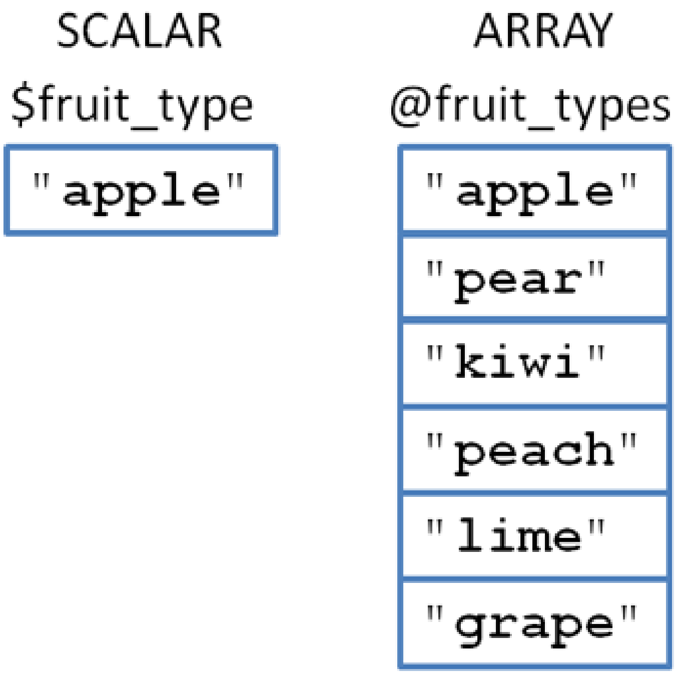
Arrays and Array Functions

Arrays

Now that you're familiar with Perl's scalar variable type, it's time to introduce you to a completely new kind of variable: the array. We know that a scalar can hold anything (a number, a string, or several things you haven't learned about yet), but it can only hold *one* thing, no more, no fewer.

The array is one of two variable types in Perl that are collectively known as *aggregates,* meaning they can hold *any* number of things. (The other aggregate is the *hash*, which we'll discuss later.)

Here's a graphical representation of our variables:



WARNING More fruit ahead. If this is making you hungry, go grab a snack immediately and come back, so you're not distracted!

The scalar in that diagram is called **$fruit_type**, and the array is called **@fruit_types**. Code is generally easier to understand if arrays are named in the plural. You'll notice that each part of the array looks remarkably similar to the scalar. That's because every part, or *element*, of an array *is* a scalar.

But because arrays are different variable types than scalars, their names start with a different punctuation character. The **@** sign was chosen because it looks like the letter **a**, which makes it a good *mnemonic device* (memory-jogger) for **a**rray.

Declaring and Using Arrays

You declare arrays just as you would scalars—using **my**:

OBSERVE: Declaring an Array

```
my @fruit_types;
```

So how do we assign arrays? An array is a collection of zero or more things. Where have we encountered something like that before? In the last lesson—lists! So the array in the example above could have been created with this code:

OBSERVE: Assigning Array Elements

```
my @fruit_types = ('apple', 'pear', 'kiwi', 'peach', 'lime', 'grape');
```

Note

You can save time and work with a shortcut for lists that are "well-behaved." Each element of a well-behaved list contains no white space (spaces or tabs). You can use the special **qw()** (quote word) function to make your code more readable:

```
my @fruit_types = qw(apple pear orange kiwi peach lime grape);
```

The **qw()** function makes your code much easier to read. The elements can't contain white space—because white space in a **qw()** operator tells Perl where one element ends and another begins. The **qw()** operator returns a list of all elements between its parentheses that have white space between them (you can put in line breaks as well). It's kind of like executing a **split()**.

Arrays are "born empty," meaning they contain zero elements. So there's no real difference between these two statements:

OBSERVE: Creating an Empty Array

```
my @fruit_types;
my @fruit_types = ();
```

Some Perl programmers will write the second form, thinking that it's somehow necessary. It isn't.

And just as you can assign a list to an array, you can assign an array to a list:

OBSERVE: Assigning an Array to a List

```
my ($fruit1, $fruit2, $fruit3) = @fruit_types;
```

If there are fewer than three elements in **@fruit_types**, then one or more of the scalars on the left side will be undefined. If there are more than three elements in **@fruit_types**, then the remaining ones will be ignored.

Now you've seen that arrays convert to and from lists in assignements, you can see how to copy one array to another:

OBSERVE: Copying One Array to Another

```
my @fruit_types = @citrus_types;
```

Let's try it in an example. Create a file called **sku.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $stock_lines = <<'END_OF_REPORT';
Inventory Stock-Keeping Units follow:
90-5825 87-9398 93-8589 40-6910
END_OF_REPORT

my ($heading, $line1) = split "\n", $stock_lines;
my @SKUs = split " ", $line1;
my ($sku1, $sku2, $sku3, $sku4) = @SKUs;
print "First SKU = $sku1; third SKU = $sku3\n";
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~$ cd perl1
cold:~/perl1$ ./sku.pl
First SKU = 90-5825; third SKU = 93-8589
cold:~/perl1$
```

We're getting a lot better at parsing text! We were able to ignore the title line of the report by putting it in **$heading** and then leaving it alone.

Arrays also interpolate in double-quoted strings. Modify sku.pl as shown below:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

my $stock_lines = <<'END_OF_REPORT';
Inventory Stock-Keeping Units follow:
90-5825 87-9398 93-8589 40-6910
END_OF_REPORT

my ($heading, $line1) = split "\n", $stock_lines;
my @SKUs = split " ", $line1;
my ($sku1, $sku2, $sku3, $sku4) = @SKUs;
print "First SKU = $sku1; third SKU = $sku3\n";
print "All SKUs: @SKUs\n";
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./sku.pl
First SKU = 90-5825; third SKU = 93-8589
All SKUs: 90-5825 87-9398 93-8589 40-6910
cold:~/perl1$
```

In the output, the array elements get printed in order, with a single space between each one.

Array Functions

Perl arrays can grow and shrink as much as you need. We know how to replace the contents of an array entirely. You don't have to predict the size of your arrays in advance either. Memory permitting, they can expand to hold one element to twenty million and back down again, effortlessly. We'll take a look at four *array functions* that will allow our arrays to expand and contract:

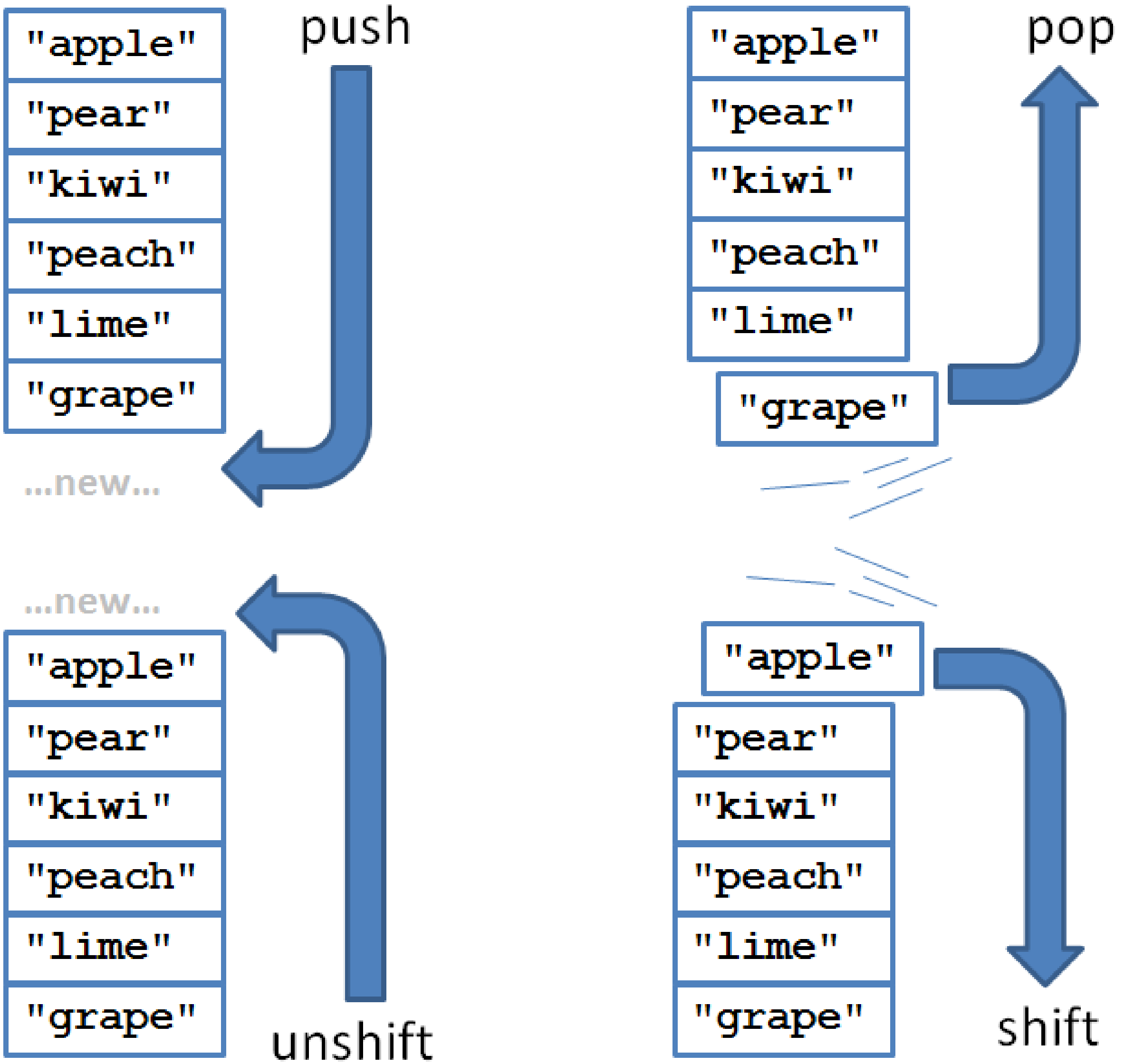
- **push @array**, **$scalar**, ...
- **$scalar** = **pop @array**
- **$scalar** = **shift @array**
- **unshift @array**, **$scalar**, ...

Of course, you can use any variable names you like; **@array** and **$scalar** are just examples. The ellipses (**...**) mean that you can have more scalars if you want; just separate them with commas (that's right, it's a list).

So, what do those functions do? You ask all the right questions!

- **push** inserts **$scalar** at the end of the array.
- **unshift** inserts **$scalar** at the beginning of the array.
- **shift** removes an element from the beginning of the array.
- **pop** removes an element from the end of the array.

When elements are added to or removed from the beginning of the array, every element in the array shifts accordingly. When elements are added to or removed from the end of the array, the array resizes accordingly. Here's a diagram to help you visualize what's taking place:



Let's try some of those functions in an example using a names database. Names come in many forms, but most databases are picky and demand a "sanitized" form of data that can be divided into a first and a last name, and possibly more. The program below uses **split()**, **shift()**, and **pop()** to locate a first and last name without counting anything that may be in between.

Create a new program called **name_find.pl** as shown below:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $name = "Charles Francis Xavier";
my @names = split ' ', $name;
my $first_name = shift @names;
my $last_name = pop @names;
print "$name is sanitized to $last_name, $first_name\n";

$name = "Hiram K. Hackenbacker";
@names = split ' ', $name;
$first_name = shift @names;
$last_name = pop @names;
print "$name is sanitized to $last_name, $first_name\n";

$name = "James Moriarty";
@names = split ' ', $name;
$first_name = shift @names;
$last_name = pop @names;
print "$name is sanitized to $last_name, $first_name\n";

$name = "Samuel Finley Breese Morse";
@names = split ' ', $name;
$first_name = shift @names;
$last_name = pop @names;
print "$name is sanitized to $last_name, $first_name\n";
```

Check Syntax and run it.

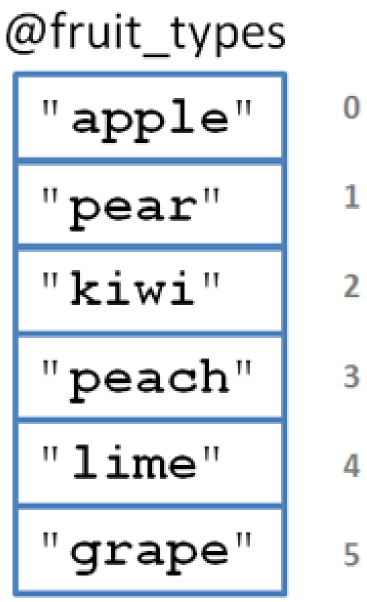
INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./name_find.pl
Charles Francis Xavier is sanitized to Xavier, Charles
Hiram K. Hackenbacker is sanitized to Hackenbacker, Hiram
James Moriarty is sanitized to Moriarty, James
Samuel Finley Breese Morse is sanitized to Morse, Samuel
cold:~/perl1$
```

The code after each different assignment to **$name** is exactly the same; don't worry about that for now. We'll learn more about that later.

Individual Array Elements

Now let's see how we can access individual elements of an array without having to assign the whole array to a list of scalars.



The numbers next to the elements of the array aren't really part of the array, they're just diagrammatic notation to help you understand that these elements are *ordered*, and can be accessed according to their location within the array. We call them *indexes* (*indices* if you want to be more formal). The index of the first element of the array is zero.

Note The first element has index 0, the second element index 1, and so on. but wouldn't it be easier to talk about array indexes if the first one was 1, the second one 2, so on? Yes, if that were the case, it would be easier to talk about them, but it would be harder to program them. The FORTRAN language made that mistake and suffered for it. If arrays started their indexing at 1, you'd be forever peppering your program with -1 and +1 expressions and forgetting which one to use when, resulting in what we call a fencepost error.

If we want to access the third element of the array of fruit types, it has the index 2. Because each element of the array is a scalar, its name starts with a dollar sign. The syntax for an individual element of an array is: dollar sign, name of array (without the @ sign), and then the index in square brackets.

Create a new program called **array_indexes.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my @fruit_types = qw(apple pear kiwi peach lime grape);
print "First fruit type = $fruit_types[0]\n";
my $index = 3;
print "Fruit type at index $index = $fruit_types[$index]\n";
shift @fruit_types;
print "First fruit type is now $fruit_types[0]\n";
print "Last fruit type is $fruit_types[-1]\n";
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./array_indexes.pl
First fruit type = apple
Fruit type at index 3 = peach
First fruit type is now pear
Last fruit type is grape
cold:~/perl1$
```

We don't often access array elements by their indexes; most of the time, we should be able to access them by other means. But still, it happens enough that you need to know how to do it.

The first element changes after a **shift()**. We don't have to assign the result of **shift()** to anything if all we want to do is throw away the first element of the array.

Okay, so what about that index of **-1**? The Perl designers decided that instead of negative indexes resulting in an error, they'd make them be useful instead. To that end, an index of **-1** refers to the *last* element of the array. An index of **-2** refers to the second-to-last element, and so on.

That's it. You are now well on your way to working effectively with arrays in Perl! Feel free to go get a smoothie or pie or whatever else you're craving after all this talk about fruit.

Your First Loop Statement

Lesson Objectives

When you complete this lesson, you will be able to:

- write a loop statement.
- use nested loops.
- use while loops.

While Loops

Good to have you back. Until now, you could only write programs that would progress in one direction: from top to bottom. The only way you could change its course was to introduce an **if** block that might cause part of the code to be skipped. That's all going to change though, as we learn about *loops*.

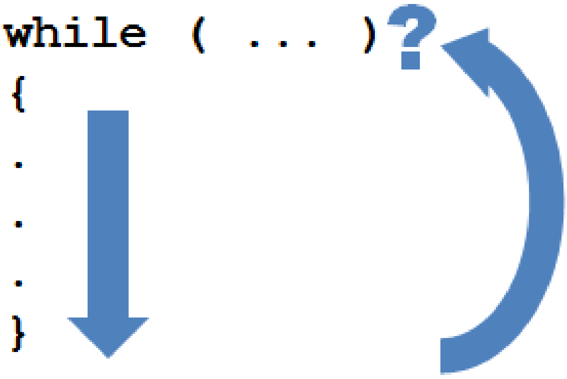
WARNING This means your program can go around in circles, potentially *never ending*. If you think that you might do that by mistake, make sure you know how to interrupt a program that's going nowhere fast. Usually you type a **Ctrl-C** in the shell window to interrupt the program.

While Loop Syntax

Remember the *condition* of an **if** statement? (I hope you do...you needed it on the last homework project!) Conditions are also used in **while** loops, like this:

OBSERVE: while statement form

```
while ( condition )
{
   code...
}
```



If the ***condition*** is true, then the ***code*** contained within the curly brackets will be executed. After executing the block, Perl evaluates the ***condition*** again, and if it's true, then Perl executes the ***code*** again. This process repeats to make the loop. If the ***condition*** is false, the first time Perl encounters it, then the block will never be executed. If it becomes false in the ***code*, the loop stops.**

Let's try an example. Create a file called **square.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $x = 0;
while ( $x < 10 )
{
  $x += 1;
  print "$x squared = ", $x**2, "\n";
}
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~$ cd perl1
cold:~/perl1$ ./square.pl
1 squared = 1
2 squared = 4
3 squared = 9
4 squared = 16
5 squared = 25
6 squared = 36
7 squared = 49
8 squared = 64
9 squared = 81
10 squared = 100
cold:~/perl1$
```

This example gives you a sense of the way **while** works in your program. Now let's mix it up a bit and add in square roots. Modify your code as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $x = 0;
while ( $x < 10 )
{
  $x += 1;
  print "$x squared = ", $x**2, "; ";
  print "Square root of $x = ", sqrt($x), "\n";
}
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./square.pl
1 squared = 1; Square root of 1 = 1
2 squared = 4; Square root of 2 = 1.4142135623731
3 squared = 9; Square root of 3 = 1.73205080756888
4 squared = 16; Square root of 4 = 2
5 squared = 25; Square root of 5 = 2.23606797749979
6 squared = 36; Square root of 6 = 2.44948974278318
7 squared = 49; Square root of 7 = 2.64575131106459
8 squared = 64; Square root of 8 = 2.82842712474619
9 squared = 81; Square root of 9 = 3
10 squared = 100; Square root of 10 = 3.16227766016838
cold:~/perl1$
```

Okay, now change **square.pl** as shown:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

my $x = 0;
while ( $x++ < 10 )
{
   ~~$x += 1;~~
   print "$x squared = ", $x**2, "; ";
   print "Square root of $x = ", sqrt($x), "\n";
}
```

and run it.

Even after we changed the code, it produced the exact same result. How can that be? The **++** operator is called the **postincrement** operator. Here's how the **++** operator works: the value of **$x++** is whatever is in **$x**, but before Perl goes on to the next statement, it adds one to **$x**.

Does that make sense? Take a look at this code:

while (**$x++ < 10**)

This code tells Perl that "if **$x** is less than **10**, then execute the block; if not, then don't execute the block. But either way, before executing the next piece of code, add one to **$x**."

This operation is called **post**incrementing because it happens *after* the statement. (There's also a **pre**increment operator, which is written like this: **++$x**.) The postincrement operator is particularly useful; you'll use it often.

Let's try another example of the **while** loop. Create a file called **order_queue.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my @inventory = qw(Pallet1 Pallet2);
my @receiving = qw(Pallet3 Pallet4 Pallet5);
my @orders = qw(Order1 Order2 Order3 Order4 Order5 Order6);

my $order;
while ( $order = shift @orders )
{
  while ( my $supply = shift @receiving )
  {
    push @inventory, $supply;
  }

  if ( my $supply = shift @inventory )
  {
    print "Fulfilling $order with $supply\n";
  }
}
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./order_queue.pl
Fulfilling Order1 with Pallet1
Fulfilling Order2 with Pallet2
Fulfilling Order3 with Pallet3
Fulfilling Order4 with Pallet4
Fulfilling Order5 with Pallet5
cold:~/perl1$
```

There are several new concepts being demonstrated at once in this program, and I'm confident you can handle them! There are *nested* **while** loops—a **while** loop (as well as an **if** block) *inside* another **while** loop. The inner blocks introduce what's known as an additional *scope* level. Levels are good (as long as you don't overdo the nesting); we'll see why later.

The conditions of each **while** loop in this code are *assignments*. That's fine—as long as you know that in Perl, the result of an assignment is the thing that got assigned.In other words, where we use an assignment as an expression—somewhere that the value gets used—the value of that expression is whatever the thing being assigned to gets set to. Take a look at the **outer while loop** code:

OBSERVE:

```
while ( $order = shift @orders )
{
}
```

This code tells Perl to **shift()** the first element of **@orders** and put it in **$order**; then if **$order** is true, execute the **while** block. When there is nothing in an array and you call **shift()**, you get back **undef**, which is false. So when there's nothing left in **@orders**, the **while** condition will be false and the block will stop being executed.

Check out this next bit of sample code:

OBSERVE: while-scoped my

```
 while  (my $supply = shift @receiving )
{
}
```

The two inner conditions contain the **my** keyword *inside* of the condition itself! That has the effect of *scoping* the variable being initialized to the block to which the condition belongs—it's as if the variable *doesn't exist* outside of that block. So, inside the outermost **while** loop, there are two *different* **$supply** variables that are *completely independent* of each other. It is as if you had written this:

OBSERVE: explicit my

```
{
  my $supply;
 while  ( $supply = shift @receiving )
  {
  }
}
```

We'll learn more about this issue of *scoping* later.

Until Loops

You can type:

OBSERVE: until

```
until  ( condition )
{
    code...
}
```

as a convenient shorthand for:

while (not)

```
while  ( ! condition )
{
    code...
}
```

By the way, you can alsotype:

unless

```
unless  ( condition )
{
    code...
}
```

as a convenient shorthand for:

if (not)

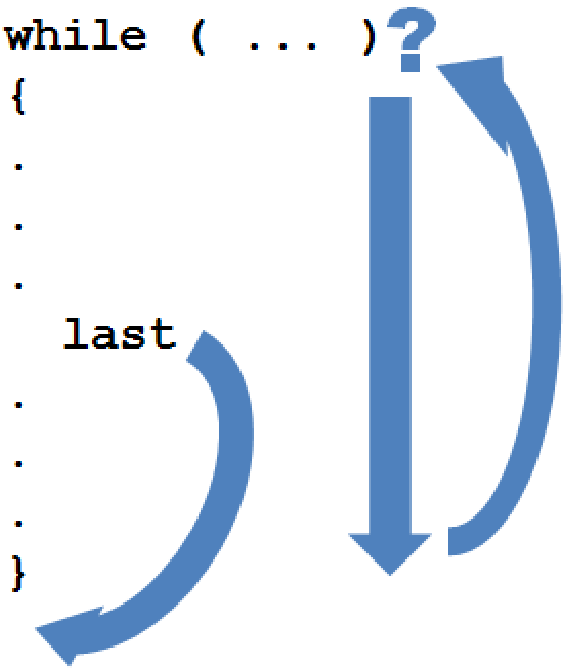
```
if  ( ! condition )
{
    code...
}
```

Pretty cool, huh?

Loop Control Statements

Now let's learn how to escape a **while** loop before it would normally stop on its own. (It's important to know how to exit an escape loop like this, just in case you create a **while** loop using the statement **while (1)**, which *never* stops on its own.)

The **last** statement (that's a statement called **last** ... not the final statement in a previous program!) means, "Exit whatever *looping* block I am currently executing as though the block had finished by itself." Note the emphasis on *looping*—this *doesn't* apply to blocks of **if** statements. If you call **last** from within an **if** block, it will exit the looping block that contains that **if** statement, or give you an error if there is no looping block anywhere around the **last** statement. I'm calling these "looping" blocks here rather than "while" blocks, because there are other kinds of looping blocks that we'll learn about later.



If Perl is working within a looping block that's located inside of another looping block when it encounters a **last** statement, the inner block is the one that is exited.

Let's look at an example. Create a file called **last.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my @instructions = qw( PUSH POP UNSHIFT QUIT NOTREACHED );
my @extra = qw( POP PUSH );
while ( my $instruction = shift @instructions )
{
  if ( $instruction eq 'QUIT' )
  {
    print "Quitting\n";
    last;
  }
  print "Executing $instruction\n";
  if ( $instruction eq 'POP' )
  {
    pop @instructions;
  }
  elsif ( $instruction eq 'SHIFT' )
  {
    shift @instructions;
  }
  elsif ( $instruction eq 'PUSH' )
  {
    push @instructions, shift @extra;
  }
  elsif ( $instruction eq 'UNSHIFT' )
  {
    unshift @instructions, shift @extra;
  }
}
```

Perl quits the **while** loop when it reads an "instruction" to quit. Can you follow the program logic and predict what items Perl will print without running it?

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./last.pl
Executing PUSH
Executing POP
Executing UNSHIFT
Executing PUSH
Quitting
cold:~/perl1$
```

We've covered a lot here. Good job staying with it! See you in the next lesson...

Formatted Printing and More Looping

Lesson Objectives

When you complete this lesson, you will be able to:

- use the built-in Perl print function.
- execute formatted printing.

Formatted Printing

Imagine you're watching one of those late-night infomercials hawking steak knives and sandwich makers. Only this time, they're saying, "Are you tired of measuring string lengths and adding spaces by hand to get output that lines up in columns? Have you ever wanted to zero-fill a number or print one to a certain number of decimal places? Well, look no further! You need **printf**, the amazing built-in Perl function! It formats numbers! It formats strings! It knows about octal, hexadecimal, and binary! But wait—there's more! You also get scientific notation, *and* left and right justification, absolutely free! Now how much would you pay?"

printf()

Fortunately, you don't have to pay anything—Perl is free! Even so, **printf** is pretty amazing, so we're going to spend some time getting familiar with it.

printf is similar to **print**—it prints its arguments—but in **printf** there's an **f** which stands for "formatted." The general syntax of **printf** is:

printf ***format***, argument, ...

This is a type of *data-driven programming*; the *format* (the first argument) is a string to be printed, but it may contain special characters that will be replaced by formatted versions of the subsequent arguments.

Let's try an example. Create a file called **printf.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

printf "I'm an ordinary string.\n";
printf "Here's a number in a field of 10 characters: <%10d>\n", 42;
printf "Here's a date: %02d/%02d/%4d\n", 7, 4, 1776;
my $PI = 3.14159265358979;
printf "PI to 3 decimal places = %.3f\n", $PI;
printf "PI to 4 decimal places = %.4f\n", $PI;
my $places = 5;
printf "PI to $places decimal places = %.*f\n", $places, $PI;
my $integer = 161;
printf "$integer in octal       = %o\n", $integer;
printf "$integer in hexadecimal = %x\n", $integer;
printf "$integer in binary      = %b\n", $integer;
printf "%10s-justified string; %-10s-justified string\n", qw<right left>;
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~$ cd perl1
cold:~/perl1$ ./printf.pl
I'm an ordinary string.
Here's a number in a field of 10 characters: <        42>
Here's a date: 07/04/1776
PI to 3 decimal places = 3.142
PI to 4 decimal places = 3.1416
PI to 5 decimal places = 3.14159
161 in octal       = 241
161 in hexadecimal = a1
161 in binary      = 10100001
right-justified string; left      -justified string
cold:~/perl1$
```

Wow! That's a lot of capability for one function, isn't it? And we've barely scratched the surface of what **printf** can do. Now, at this point you might consider rushing to **perldoc -f printf** or http://perldoc.perl.org/functions/print.html to learn all the details you can about **printf**. But you won't find them there. There's another function called **sprintf** (which along with **printf** may be familiar to you if you're already a C programmer). **sprintf** is just like **printf** except that instead of printing the formatted string, it returns it. Check out this **sprintf** example:

OBSERVE:

```
$title = sprintf "Report of store #%4d for %02d/%02d\n",
$storenum, $month, $day;
```

Information about the elements that can go into the *format* argument is identical for both **sprintf** and **printf**. Rather than write the same information twice in the documentation, the developers of Perl included the format argument information once, in **sprintf**. You can find out all about those things beginning with % (called **conversions**) through **perldoc -f sprintf** or http://perldoc.perl.org/functions/sprintf.html.

By the way, I snuck in one extra thing in **printf.pl**; did you spot it? The **qw** operator isn't using parentheses to delimit its argument **right left**; it uses angle brackets instead. You can use any "paired" group of delimiting characters you want instead of parentheses, such as angle brackets (**<>**), square brackets (**[]**), or curly brackets (**{}**).

The sheer range of things that **printf** can do means that the documentation (for **sprintf**) is pretty long. I'll summarize some of the most useful parts.

Everything in the string that is the first argument to **printf** will get printed out just as it is, with the exception of **conversions** beginning with a % character. Because these conversions are done at run-time, you can supply that first argument in any number of ways: as a single-quoted string, a double-quoted string, or a scalar. It's common to use a double-quoted string so that we can put a newline where we need it using **\n**. When you do use a double-quoted string, scalars and arrays will interpolate in it, so make sure you don't use double-quoted strings if your scalars and arrays might contain % signs that you don't want. Usually, if you want to output a string that's in a scalar, you'll use the **%s** conversion:

OBSERVE:

```
printf "Here's a string: $scalar";       # Not recommended
printf "Here's a string: %s", $scalar;   # Preferred
```

Here are the most common conversions:

- %s: string
- %d: integer
- %f: floating-point number
- %o, %x, %b: octal, hexadecimal, and binary, respectively
- %%: a percent character

Each one of those will format the output using exactly the space required, but you can change that. By putting a number between the % and the letter, you can pad the conversion result with spaces on the left so that it is right-justified in a field of the specified number of characters. If you put a minus sign in front of the number, it left-justifies:

OBSERVE: printf justification

```
printf "|%8d|\n", 123;     # prints |     123|
printf "|%-8d|\n", 123;    # prints |123     |
```

The vertical bars in these examples are there to indicate the boundaries of the resulting fields. Width and justification works for all of the conversions listed above. If you specify a width that is actually narrower than what you're printing, Perl ignores the width. There is no "center" justification.

For the **%d** conversion, you can prefix a width with a zero to get zero fill instead of space padding:

OBSERVE: zero filling

```
printf "|%012d|\n", 234523;    # prints |000000234523|
```

For the **%f** conversion, you usually specify a *precision* in addition to a width; the precision is given as a number after a period following the width. Take a look at this example of **floating-point precision**:

OBSERVE: floating-point precision

```
printf "|%12.6f|\n", 234.523;    # prints |  234.523000|
```

The default precision is 6. You can omit the width and just specify the number of digits to print after the decimal point. Here's an example that uses **floating-point decimal places**:

OBSERVE: floating-point decimal places

```
printf "|%.2f|\n", 234.523;    # prints |234.52|
```

For the **%s** conversion, precision is interpreted as a maximum field width; the string will be truncated from the right as necessary to fit:

OBSERVE: string precision

```
printf "|%.6s|\n", "abcdefghi";     # prints |abcdef|
printf "|%8.6s|\n", "abcdefghi";    # prints |  abcdef|
```

Scientific Notation

Sometimes you need to deal in numbers that are so big (or so small) that the number of digits makes them hard to read. Scientists have to do this all the time, which is why they invented *scientific notation*, a way of expressing those numbers with a floating point number in the range 1 to (just under) 10 and a power of ten to multiply the result by. So 299792458 (speed of light in meters per second) can be represented as 2.99792458 times 10 to the power 8, which is written as **2.99792458E+08**, and 0.000000000000000000000000000091093829 (mass of an electron in grams) can be written as **9.10938291E-28**.

Perl can use those scientific notation numbers as program literals. A plus sign after the E is optional; so are leading zeros in the digits following the E (called the *exponent*), and the E can be upper or lower case. You can also print them out, using **printf**:

OBSERVE: scientific notation

```
printf "Light-year in meters: %e", 2.99792458E8 * 60 * 60 * 24 * 365.24;   # prin
ts Light-year in meters: 9.460471e+15
```

You can see there a literal scientific number (speed of light in meters per second), and the output from the **%e** conversion is in scientific notation. There is also a friendly **%g** conversion that will use **%e** if the number is very big (or small) and **%f** otherwise.

We'll do another **printf** example shortly, right after we've learned another piece of syntax.

foreach Loops

Here's the general form of a **foreach** loop:

OBSERVE: foreach statement form

```
foreach variable( list )
{
  code...
}
```

The ***variable*** will be set in turn, to each of the elements of the ***list***, and the ***code*** will be executed each time. Usually (but not always, as you'll see shortly), you'll reference the variable in the code so that the code does something different each time.

Here are a couple of examples:

OBSERVE:

```
foreach $fruit( @fruits )
{
  print "Found a $fruit\n";
}
```

OBSERVE:

```
# $lines contains lines that look like:
# SKU cost cost cost ...
foreach my $line( split "\n", $lines )
{
  my ($sku, @costs) = split " ", $line;
  my $total = 0;
  print "SKU $sku has total cost ";
  foreach my $cost( @costs )
  {
    $total += $cost;
  }
  print "$total\n";
}
```

The .. Range Operator

Sometimes you'll want to execute the same piece of code a certain number of times without regard to any loop variable. In such a case, you could write code that looks like this:

OBSERVE: repetitious Loop

```
foreach my $whatever ( 1, 2, 3 )
{
  print "I tell you three times\n";
}
```

If you wanted to go around the loop a hundred times, you'd have to do a lot of typing! This is where the *range operator* comes in handy:

OBSERVE: range operator

```
number1 .. number2
```

An expression of this form evaluates to the *list* of all the numbers in the range *number1* to *number2*, inclusive. So we could rewrite the loop above as:

OBSERVE:

```
foreach my $whatever ( 1 .. 3 )
{
  print "I tell you three times\n";
}
```

Now, if we want to change our code to make it go around the loop a hundred times, it won't be a problem. It's also less difficult to set parameters:

OBSERVE:

```
my $count = 3;
foreach my $whatever ( 1 .. $count )
{
  print "I tell you $count times\n";
}
```

Now let's use **foreach** in a **printf** example. You may remember this program from lesson 6:

OBSERVE: store.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $lines = <<'END_OF_REPORT';
0.95  300   White Peaches
1.45  120   California Avocados
5.50   10   Durien
0.40  700   Spartan Apples
END_OF_REPORT

my ($peaches_sold, $avocados_sold, $durien_sold, $apples_sold) = (12, 6, -1, 24)
;  # One durien returned... too smelly
my ($line1, $line2, $line3, $line4) = split "\n", $lines;
my ($cost, $quantity, $item) = split " ", $line1, 3;
$quantity -= $peaches_sold;
$line1 = join " ", $cost, $quantity, $item;
print "$line1\n";
($cost, $quantity, $item) = split " ", $line2, 3;
$quantity -= $avocados_sold;
$line2 = join " ", $cost, $quantity, $item;
print "$line2\n";($cost, $quantity, $item) = split " ", $line3, 3;
$quantity -= $durien_sold;
$line3 = join " ", $cost, $quantity, $item;
print "$line3\n";
($cost, $quantity, $item) = split " ", $line4, 3;
$quantity -= $apples_sold;
$line4 = join " ", $cost, $quantity, $item;
print "$line4\n";
```

That's kind of hard to read, isn't it? Let's tidy it up, and make it clearer and more extensible. Create a new file as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $lines = <<'END_OF_REPORT';
 0.95   300   White Peaches
 1.45   120   California Avocados
10.50    10   Durien
 0.40  1500   Spartan Apples
END_OF_REPORT

my @sold = (12, 6, -1, 24);  # One durien returned... too smelly
foreach my $line ( split "\n", $lines )
{
  my ($cost, $quantity, $item) = split " ", $line, 3;
  $quantity -= shift @sold;
  printf "%5.2f %6d %s\n", $cost, $quantity, $item;
}
```

💾 it in your **/perl1** folder as **store_report.pl** and run it.

INTERACTIVE TERMINAL SESSION:

```

cold:~/perl1$ ./store_report.pl
 0.95    288 White Peaches
 1.45    114 California Avocados
10.50     11 Durien
 0.40   1476 Spartan Apples
cold:~/perl1$

```

💾 Save and run it. A few of the numbers have been changed which makes it easier to see what **printf** is actually doing for us. The output report is lined up in columns just like the input.

By the way, some of the input lines in this program begin with spaces to allow the *costs* to line up, yet **split()** still does the right thing. That's because **split()** ignores any whitespace at the beginning of the string when its splitting on whitespace. That's one of those really handy Perl behaviors that you may not even notice.

If you're slightly uncomfortable with the fact that the program we've got depends on the input and the array **@sold** being in the same order, good! That's how a programmer should think! We'll find out what to do about that dilemma later.

For now, get busy with the lesson's assignments and press on!

Perl's Contextual Behavior

Lesson Objectives

When you complete this lesson, you will be able to:

- write Perl code in the proper context.
- use the concatenation operator to put arguments in scalar context.

Context

"The skill of writing is to create a context in which other people can think."
-*Edwin Schlossberg*

Scalar and List Context

Remember a few lessons ago when I referred to the notion of something "expecting" a list and said, "We have an entire lesson dedicated to the intricacies of lists later")? Well, "later" is now! One of the fundamental concepts in Perl is *context*. The definitive book on the language, Programming Perl, claims that Perl programmers will be miserable until they learn it. Well, we certainly don't want that to happen to us, so let's get busy learning! In basic terms, context in Perl is defined like this:

> Any expression can be interpreted in several different ways; the way that is chosen depends on how the expression is being used.

This concept will become much more clear to you after we go over some examples. Let's review our code from insect_sales.pl:

OBSERVE:

```
my $date = localtime().
```

The result of **localtime()** is being assigned to a *scalar*, so we say that the assignment places **localtime()** in **scalar context**. And **localtime()** in a scalar context returns a string representing the current date and time. It would look something like this: **'Mon Aug 10 10:04:49 2009'**.

But *instead* of assigning **localtime()** to a scalar, we could assign it to an array. Let's try an example. Create a new file as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my @parts = localtime();
my $count = 0;
foreach my $part ( @parts )
{
  print "Element $count = $part\n";
  $count++;
}
```

Check Syntax Save it in your **/perl1** folder as **localtime.pl** and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~$ cd perl1
cold:~/perl1$ ./localtime.pl
Element 0 = 0
Element 1 = 50
Element 2 = 18
Element 3 = 7
Element 4 = 11
Element 5 = 109
Element 6 = 1
Element 7 = 340
Element 8 = 0
cold:~/perl1$
```

Before we go any further, take a good look at the code and see if you can figure out what it means.

Also, observe that the **++** operator is used for its side effect of adding one to its operand, without looking at its value. This is a common use of the **++** operator.

Now, look at **perldoc -f localtime** or http://perldoc.perl.org/functions/localtime.html. **localtime()** returns a list of numbers representing elements of the current date and time, so you can access each element—the day, the month, etc.—individually.

The important lesson here is that **localtime()** behaves differently depending upon the context. There are two important contexts to learn:

- Scalar
- List

And any expression can be placed into either context by putting it somewhere that "expects" either a scalar or a list. For example, **print** expects a list of arguments, so if you **print localtime()**, all of those numerical date/time elements will be printed out, one after the other, without spaces in between them. If you wanted to print the scalar context value of **localtime()** without having to assign it to a scalar variable first, you'd look for an expression that would put part of itself in scalar context. For example, string concatenation:

OBSERVE:

```
$x . $y
```

Perl only concatenates scalars, so the concatenation operator puts each of its arguments in scalar context:

OBSERVE:

```
print localtime() .  "\n";
```

If you just want to print out a readable time in a hurry, that's a fine way of doing it, although it might be challenging for others to read your code.

There's no way of predicting how **localtime()** will behave in one context based on its behavior in another context. The people who wrote **localtime()** made good, useful choices, but there is no general rule for the way a function will behave in a list context, even when you know its behavior in a scalar context or vice versa.

A scalar variable (say, **$x**) in a scalar context, yields that variable. A scalar variable in a list context (say, **@y** = **$x**) yields a list containing only that variable. An array in a list context, for instance:

OBSERVE:

```
($x, $y) = @z;
```

...yields the list of elements in the array. In this particular case, we know that the array is in list context because it's being assigned to a list—a set of scalars separated by commas *inside parentheses*.

By the way, just because a code snippet doesn't declare its variables with **my**, doesn't mean you shouldn't.

Sometimes I'll leave out the snippet of code that declares variables just to make the code easier for you to read overall, but when you write a program, you need to include the **my** declarations. **use strict** won't let you get away with leaving out **my**, and you're always using **use strict**, aren't you?

So, what about an array in a scalar context? Let's try that! Create a file called **scalar_array.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my @array = qw(apple pear loganberry starfruit durien kumquat);
my $whatever = @array;
print "Array in scalar context = $whatever\n";
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./scalar_array.pl
Array in scalar context = 6
cold:~/perl1$
```

An array in a scalar context—used wherever Perl would expect a scalar—evaluates as the number of elements in the array. Here are some ways of using scalar context to get to that:

- **$count** = **@array**
- **0 + @array**
- **scalar**(**@array**)
- **while** (**@array**)

In the third item on the list, the **scalar()** function is a Perl *built-in* that forces its argument into scalar context. (There's no **list()** function; you won't need it.)

Now take a look at the last item. How does that one work? The condition of a **while()** statement must be either true or false, so it's evaluated in *Boolean context*, which is for all practical purposes the same as scalar context. And whatever comes between the parentheses is going to be put in scalar context. By the way, this example demonstrates another important difference between the **while** and **foreach** loops (your homework from the previous lesson may come to mind): **while** puts the contents of the parentheses—its condition—into scalar context, and **foreach** puts the contents of its parentheses—its arguments—into list context.)

Let's incorporate that into one of our earlier examples. Create a new file called **name_finder.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my @names = ("Charles Francis Xavier", "Hiram K. Hackenbacker", "James Moriarty"
, "Samuel Finley Breese Morse");
while ( @names )
{
  my $name = shift @names;
  my @parts = split ' ', $name;
  my $first_name = shift @parts;
  my $last_name = pop @parts;
  print "$name is sanitized to $last_name, $first_name\n";
}
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./name_finder.pl
Charles Francis Xavier is sanitized to Xavier, Charles
Hiram K. Hackenbacker is sanitized to Hackenbacker, Hiram
James Moriarty is sanitized to Moriarty, James
Samuel Finley Breese Morse is sanitized to Morse, Samuel
cold:~/perl1$
```

This version of the program will no longer stop when it hits an element of the array **@names** that happens to be false—not a likely possibility with names, but if you're iterating over an array that might contain a false element, you'll want to take this approach.

Now, modify the program as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my @names = ("Charles Francis Xavier", "Hiram K. Hackenbacker", "James Moriarty"
, "Samuel Finley Breese Morse");
foreach my $name ( @names )
{
  ~~my $name = shift @names;~~
  my @parts = split ' ', $name;
  my $first_name = shift @parts;
  my $last_name = pop @parts;
  print "$name is sanitized to $last_name, $first_name\n";
}
```

and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./name_finder.pl
Charles Francis Xavier is sanitized to Xavier, Charles
Hiram K. Hackenbacker is sanitized to Hackenbacker, Hiram
James Moriarty is sanitized to Moriarty, James
Samuel Finley Breese Morse is sanitized to Morse, Samuel
cold:~/perl1$
```

This version of the code does the same thing as the last one and demonstrates the array being placed in list context. Because **shift** isn't used, the array is not modified, and because **foreach** specifies a block iteration variable (**$name**), the code is slightly shorter. This version is arguably better than the previous version.

Any function in Perl may return different results in list context and scalar context, and many do. When you learn to write your own functions (in the next lesson), you'll be able to tell whether something was called in list context or scalar context by calling the **wantarray** built-in function.

Keep in mind that when you assign to a list of scalars, they should be separated by commas and surrounded by parentheses:

(**$x**, **$y**, **$z**) = ...

Doing that puts the right side of the **=** sign into list context. So, what's the difference between these two statements?:

- **$x** = ...
- (**$x**) = ...

The first statement puts the right side of the **=** sign into scalar context; the second puts it into list context. The list to which items are being assigned may have only one element in it, but it's still a list, as opposed to a scalar that is not within a list (like in the first statement).

Note By the way, the authors of "Programming Perl" weren't kidding when they mentioned the misery that might follow when a programmer doesn't quite understand context. In fact, one person has a police record as a result of not understanding the difference between scalar and list context! Read this Slashdot article, a real-life cautionary tale!

This lesson is a little shorter than previous ones, because the homework is a little more labor intensive. There are many subtleties to understand about lists and context in Perl, so hopefully you'll find yourself exploring. If you get confused, stop and ask yourself whether you're wrestling with code that you might really want to use. If it seems potentially useful, then keep going as far as you like!

By now, you're getting a solid understanding of the fundamentals of Perl and you know quite a few things that casual Perl programmers probably don't. You're doing great! Keep going!

Creating Reusable Code

Lesson Objectives

When you complete this lesson, you will be able to:

- create reusable code.
- use subroutines.
- define and use digraphs.

Subroutines

One of the guiding principles of computer program design has been distilled into this simple phrase: **D**on't **R**epeat **Y**ourself (**DRY**).*

In other words, don't write the same code twice if it's doing the same thing each time. Instead, put that code in one place and call it from the two (or more) places that you need it.

So, how do you do that? Great question. Consider the **printf()** function. Imagine for now that it doesn't exist, but you need its functionality. You write some code that will format a number in a fixed-width field by turning the number into a string, calling the **length()** function, printing out the number of spaces you need, and so on.

Later, you need to do that again in a different part of your program to a different number. Do you write the same code out again? Of course not—but you'd be amazed at how often people do just that! Fortunately, we are not those people. Instead, we'll use the **printf()** function to help us call our code whenever we need it.

We create this kind of reusable code using **subroutines**.

* This concept first appeared in *The Pragmatic Programmer*, by Andy Hunt and Dave Thomas (Addison-Wesley, 1999).

Subroutines

A subroutine is the basic building block of reusable code in Perl. If you've used the equivalent of subroutines in other languages, you may be familiar with terms like *functions* and *procedures*. We don't use the term "procedures" in Perl programming, and when we talk about "functions," we're generally referring to built-in functions. Technically there's no difference in behavior between functions and subroutines though. Also, subroutines in Perl always refer to code written by the programmer, not a part of Perl itself.

If you have programmed things like subroutines in other languages, put that knowledge aside for this lesson. Perl subroutines work much more simply than subroutines in other languages, and several of the features you may be used to using don't exist in Perl.

Here's how you define a subroutine in Perl:

OBSERVE:

```
sub my_sub
{
  # Body of subroutine
}
```

This subroutine is named **my_sub**. Here's how you'd call it from somewhere in your Perl program:

OBSERVE:

```
my_sub( ... );
```

Pretty cool, huh? The ellipsis (**...**) indicates that "any arguments go here."

When you run a Perl program, Perl actually reads your entire program before doing anything—this is called the *compilation* phase (it happens much faster in Perl than it does in most other compiled languages). Then Perl runs your program. When Perl encounters the definition of a subroutine during compilation, it tucks it away and remembers it. When Perl runs your program, it's as though the subroutine definitions aren't there—

they don't get executed until they are called from a part of the program that's not in a subroutine definition. (Subroutines that aren't called at all are referred to as *orphaned code* and might as well be left out of your programs.)

Before we go on, let's go over few guidelines to make sure that you'll follow best practices and don't become confused by Perl's flexible subroutines syntax options:

- Define your subroutines after the main code (this makes the code more readable).
- Call your subroutines with parentheses, even if there are no arguments between the parentheses.
- Don't give a subroutine the same name as a Perl built-in function.

That last rule might seem a bit difficult to follow—I mean, how can you remember the names of all the Perl built-ins? And what are they anyway?

Well, you can look them over them if you execute **perldoc perlfunc** (or http://perldoc.perl.org/perlfunc.html). It's a big page. In fact, this is the source of [perldoc -f] information. You don't have to read the whole thing; all of the functions are listed in groups at the top of the page.

Still, implementing that third rule looks like a lot of work. Fortunately, we have another rule to add that will eliminate a big chunk of that work:

- Include an underscore in your subroutine name.

The presence of the underscore *guarantees* that your subroutine will not have the same name as a built-in, because none of the built-ins have underscores in their names.

Believe it or not, after enough programming in Perl, you'll know the built-in function names well enough to be able to skip the underscore rule, but until then, use them. Just think of a brief description of your subroutine's task, write it in terms of a few words, and then put underscores between the words, like this:

- trim_strings()
- add_costs()
- ping_servers()
- launch_nuclear_missiles()

(If you find yourself writing a Perl subroutine with that last name, please email me before putting it into production. I'd like to check your code.)

Let's explore how subroutines work with an example. Create a new file called **magnitude.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

foreach my $bignum (qw(7893420000 42 264340000000 3460000000000 96450000 24300))
{
  my $nice_form = number_suffix($bignum);
  print "$bignum = $nice_form\n";
}

sub number_suffix
{
  my $n = shift;

  my $suffix = '';
  my @threshold = ( K => 1E3, M => 1E6, G => 1E9, T => 1E12 );
  while ( my $divisor = pop @threshold )
  {
    my $l = pop @threshold;
    unless ( $n >= $divisor )
    {
      next;
    }
    $n /= $divisor;
    $suffix = $l;
    last;
  }
  return "$n $suffix";
}
```

(Are you wondering what the **next** statement is? It was in the homework for an earlier lesson. Run [perldoc perlsyn] if you need a reminder.)

Check Syntax and run it.

INTERACTIVE SESSION:

```
cold:~$ cd perl1
cold:~/perl1$ ./magnitude.pl
7893420000 = 7.89342 G
42 = 42
264340000000 = 264.34 G
3460000000000 = 3.46 T
96450000 = 96.45 M
24300 = 24.3 K
cold:~/perl1$
```

Our output tells us that our program is turning big numbers into more manageable numbers by adding the common suffixes of 'K' for kilo (thousand), 'M' for mega (million), 'G' for giga (billion), and 'T' for tera (thousand billion).

In addition to subroutines, this program introduces us to the term **digraph**. "Digraph" refers to any pair of characters together that means something special. For example, **=>** means the same thing as a comma, except that Perl pretends there are single quotation marks around the word to the left of it, provided that word is "well-behaved." For our purposes that means it obeys the same rules of syntax as something that would be legal as the identified part of a variable (coming after the **$** for a scalar or the **@** of an array name). The Perl term for that digraph is the **fat comma**, and it's an example of what we call **syntactic sugar**, that is, something that isn't absolutely necessary (because we could do it with existing syntax), but it makes reading the program easier. If I didn't use the digraph in that code, this line:

OBSERVE: with fat comma

```
my @threshold = ( K => 1E3, M => 1E6, G => 1E9, T => 1E12 );
```

...would read:

OBSERVE: without fat comma

```
my @threshold = ( 'K', 1E3, 'M', 1E6, 'G', 1E9, 'T', 1E12 );
```

Which version do you prefer? Why? (These questions are designed to get you thinking about code *readability* and *maintainability*.)

Okay, now let's look at how the subroutine works here. We're *calling* the routine **number_suffix** with one argument (**$bignum**) each time through the **foreach** loop. We're getting back a string that is suitable for printing (contains the scaled number and its suffix). So what happens in between?

The code in the body of **number_suffix** is executed, starting with the line:

OBSERVE: subroutine argument

```
my $n = shift;
```

Now, we know what **shift** does—but what array are we shifting? We're shifting the special array:

@_

That's an underscore. Within the definition of a subroutine in Perl, the default array in **pop** and **shift** is **@_**, so:

OBSERVE: subroutine argument

```
my $n = shift;
```

...is exactly the same as:

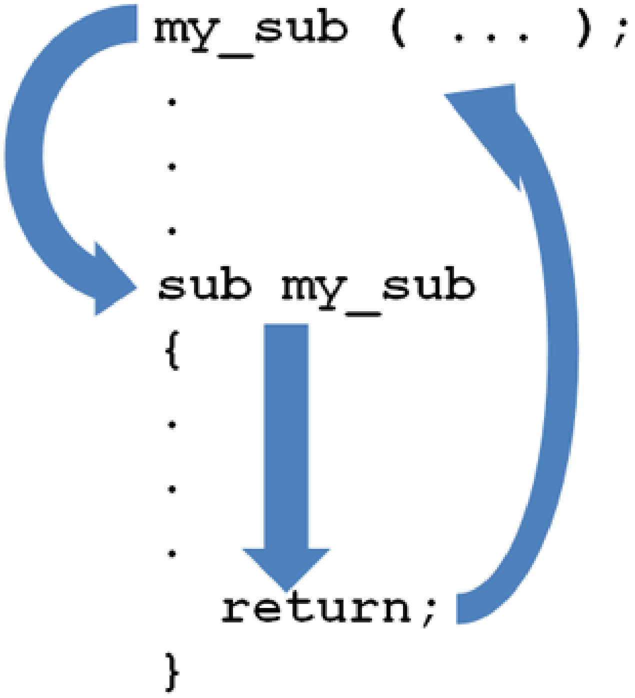
OBSERVE: subroutine argument

```
my $n = shift @_;
```

...when it's inside a subroutine definition. The universal idiom (common coding practice) is to leave out the **@_** in this case, so that's what we've done.

So, what is the **@_** array? It's the array in which the actual arguments to the subroutine are passed. In this case, **@_** will contain one element, **$bignum** (technically, it contains an *alias* to **$bignum**), and the **shift** line takes that element out of **@_** and copies it into **$n**. Because **$n** is *declared* with **my** inside the body of the subroutine, only the code in that subroutine body can reference that **$n**.

Finally, the body of the subroutine calculates the result. When it comes time to return a result—because we do assign the result of **number_suffix** to something (**$nice_form**)—we use the special keyword **return** to mean, "return from the subroutine now and pass back whatever comes after **return** as the result."



If you've programmed subroutines (or functions, or procedures) in other languages before, you may be pondering questions about prototypes, and return signatures, and pass-by-reference, and variadic argument lists. Don't worry about that stuff; Perl doesn't use or need them. Also, it's possible in Perl to overwrite an argument passed to a subroutine—called an INOUT parameter in some languages—but it's bad practice, so don't even try. And there you have it. You've just learned everything you need to know about subroutines in Perl. Nice, huh?

Let's keep going! Perl will let you return a list by entering your list after **return**. And Perl has no problem allowing you to assign variables you're passing. It looks like this:

OBSERVE: result assignment

```
($x, $y) = cube_it( $x, $y );
```

This code is preferable to overwriting arguments in place, because in this code, the action is made clear to the caller.

There is *no* validation of parameters—you can pass zero arguments to a subroutine on one call and five thousand arguments on the next, and Perl won't complain. If you don't want people to call your subroutine in a particular way, it's up to you to inspect the size and contents of **@_** and make your own complaint. Here's how we might do that in our example. Edit **magnitude.pl** as shown:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

foreach my $bignum (qw(7893420000 42 264340000000 3460000000000 96450000 24300))
{
  my $nice_form = number_suffix($bignum);
  print "$bignum = $nice_form\n";
}

sub number_suffix
{
  unless ( @_)
  {
    die "number_suffix needs an argument\n";
  }
  my $n = shift;

  my $suffix = '';
  my @threshold = ( K => 1E3, M => 1E6, G => 1E9, T => 1E12 );
  while ( my $divisor = pop @threshold )
  {
    my $l = pop @threshold;
    unless ( $n >= $divisor )
    {
      next;
    }
    $n /= $divisor;
    $suffix = $l;
    last;
  }
  return "$n $suffix";
}
```

and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./magnitude.pl
7893420000 = 7.89342 G
42 = 42
264340000000 = 264.34 G
3460000000000 = 3.46 T
96450000 = 96.45 M
24300 = 24.3 K
cold:~/perl1$
```

We put @_ in scalar context there.

The other changes we made show once again that TMTOWTDI (**T**here's **M**ore **T**han **O**ne **W**ay **T**o **D**o **I**t)!

Our next topic is fairly advanced, but I know after our discussion of the importance of context, you're going to want to be able to identify the context in which a subroutine is being called. Work this next example to find out how to do that. Modify **magnitude.pl** as follows:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

~~foreach my $bignum (qw(7893420000 42 264340000000 3460000000000 96450000 24300))~~
~~{~~
~~  my $nice_form = number_suffix($bignum);~~
~~  print "$bignum = $nice_form\n";~~
~~}~~
my $bignum = 7893420000;
my $nice_form = number_suffix($bignum);
print "$bignum = $nice_form\n";

$bignum = 264340000000;
print "$bignum = ", number_suffix($bignum), "\n";

$bignum = 3460000000000;
number_suffix($bignum);

sub number_suffix
{
  unless ( @_ )
  {
    die "number_suffix needs an argument\n";
  }
  my $n = shift;

  my $suffix = '';
  my @threshold = ( K => 1E3, M => 1E6, G => 1E9, T => 1E12 );
  while ( my $divisor = pop @threshold )
  {
    my $l = pop @threshold;
    unless ( $n >= $divisor )
    {
      next;
    }
    $n /= $divisor;
    $suffix = $l;
    last;
  }

  if ( wantarray )
  {
    return ($n, $suffix);
  }
  elsif ( defined(wantarray) )
  {
    return "$n $suffix";
  }
  else
  {
    print "$n $suffix\n";
  }
  ~~return "$n $suffix";~~
}
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./magnitude.pl
7893420000 = 7.89342 G
264340000000 = 264.34G
3.46 T
cold:~/perl1$
```

We're calling **number_suffix** three different ways: first in scalar context (because the result is assigned to **$nice_form**); second in list context (because it appears in the argument *list* of the **print** function); and third in a new context to you, *void* context (because there is *no* assignment to anything). We can use **wantarray()** to figure out which context is in use (and yes, it should have been called **wantlist()**, but we're stuck with it now). **wantarray()** returns a true value when we're in list context, a false (but defined) value in scalar context, and the undefined value in void context. We test for the last case using the built-in function **defined()**, which returns false if its argument is undefined. Make sense?

Be sure you understand how your code works; in particular, why there is a space between the number and the 'G' suffix in the first line of output, but not the second.

The void context case of this subroutine actually violates a number of good practices in code design, but we let it slide this time, just for the sake of example. Usually we don't even test for void context when writing subroutines, but it's good practice to test for scalar versus list context if you think your subroutine might return different things in each context.

A few important notes concerning the results of your subroutine queries:

- You don't have to use the results that a subroutine returns; you can ignore them by calling it in void context. (If you don't want someone to call in void context, now you know how to test for it!)
- Subroutines don't have to contain a **return** statement, but all subroutines return something. If they come to the end of their definition without encountering a **return** statement, then they will return the value of the last expression that was evaluated in any statement. If your subroutine will only be called in void context, none of this matters.
- If you enter an empty **return;** (no arguments) in a subroutine, it will return the undefined value in a scalar context and the empty list in a list context. This can be really useful in well-designed subroutines.

Let's do one more example so you can see the common practice for reading multiple arguments. Create a new file called **make_metric.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $GALLONS_TO_LITERS = 3.87;
my $POUNDS_TO_KG      = 0.45;

my $quantity = 0.4;
my $liters = make_metric( $quantity, "gallons" );
print "$quantity gallons = $liters liters\n";

my $kg = make_metric( $quantity, "pounds" );
print "$quantity lbs = $kg kg\n";


sub make_metric
{
  my ($quantity, $type) = @_;

  if ( $type eq 'gallons' )
  {
    return $quantity * $GALLONS_TO_LITERS;
  }
  elsif ( $type eq 'pounds' )
  {
    return $quantity * $POUNDS_TO_KG;
  }
  else
  {
    die "I don't know what to do with $type\n";
  }
}
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./make_metric.pl
0.4 gallons = 1.548 liters
0.4 lbs = 0.18 kg
cold:~/perl1$
```

A few more characteristics of subroutines to take note of:

- The **$quantity** inside **make_metric** is different from the **$quantity** earlier in the program, because it is declared in the subroutine with **my**. After that declaration (which also initializes it to the first element of **@_**), the subroutine **make_metric** cannot even see the **$quantity** in the outer program—it has no way to refer to it.
- Unlike our earlier example, this program does not take anything out of **@_**—it could go back to the array again if it wanted to, but the universal best practice is to copy the arguments out in the first line of the subroutine and never refer to **@_** again. The alternative is to **shift** the elements out of **@_** into variables that are local to the subroutine—as we did with **magnitude.pl**—but we usually don't take that approach if we expect to be passed more than one argument.
- This program follows a best practice of declaring "magic numbers" (the conversion factors) up at the top of the program by putting them in variables with descriptive names. For more best practices, see the book Perl Best Practices by Damian Conway (O'Reilly, 2005).

Subroutines are handy and will save us a lot of time. Experiment and play around with what you've learned to make sure you can take full advantage of them. See you in the next lesson!

Data Mapping with Hashes

Lesson Objectives

When you complete this lesson, you will be able to:

- use the hash data type.
- map data using hashes.
-

Hashes

We're about three-quarters of the way through the course and you already have the skills to create some really useful programs. But there's still lots more to learn about Perl! Let's go over one of the most important features of the language: **hashes**.

Before the invention of Perl, most languages didn't include the hash data type, but it became apparent right away that this type was extremely useful. Nearly every language invented after Perl includes hashes. And virtually every program you'll ever write to perform a real task will use this data type.

Getting Started with Hashes

If you've worked in other languages (or earlier versions of Perl), you might already understand the concept of a key => value structure as an *associative array* or *dictionary*. In current Perl, it's called a hash.

A hash contains *elements*, each of which consists of a pair of scalars: a *key* and a *value*. (Technically, the *key* is a really a string. You'll learn about the distinctions between them in later Perl courses.)

A hash (like an array) is an *aggregate*: it may contain zero or more elements. But unlike an array, there is *no order* to the elements within a hash.

You might think that's weird, especially if you know that computer memory is arranged in linear order; surely the elements *have* to be in *some* order, right? Well, in fact there is a kind of order to them, but it won't make any sense to you, it isn't useful for anything, and it can change without warning. We'll just let it go at that and avoid unnecessary confusion!

When you store an element in a hash, you must specify the key and the value. If the element with that key already exists, the value is overwritten; if it does not exist, then it is created. When you retrieve an element, you specify the key and you get back the value. So, the keys of a hash are all unique, but the values do not need to be unique.

Okay, let's see how this works with a return to Echidna Eric's Insect Emporium! Create a file called **emporium.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my (%count, %cost);
$count{'ants'}        = 47_000;
$count{'fleas'}       = 240_000;
$count{'beetles'}     = 520;
$count{'fruit flies'} = 1_500_000;

$cost{'ants'}        = 0.10;
$cost{'fleas'}       = 0.04;
$cost{'beetles'}     = 0.02;
$cost{'fruit flies'} = 0.001;

my $insect = 'ants';
print "Value of $insect on hand = ", $count{$insect} * $cost{$insect}, "\n";
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~$ cd perl1
cold:~/perl1$ ./emporium.pl
Value of ants on hand = 4700
cold:~/perl1$
```

Check to make sure you get the correct result. Now try changing the value of **$insect** to **'fleas'**, **'beetles'**, and **'fruit flies'** in turn and see if you get the right results for those values too. Finally, try setting **$insect** to a different value of your choice.

Okay, now let's dissect this script:

OBSERVE: Hash Declaration

```
my (%count, %cost);
```

This line declares two hashes. Because it's a new data type, the hash begins with a new punctuation character, the percent sign (**%**). We won't use **%** very often because most of the time our programs will refer to individual hash elements rather than the hash that contains them. But we need to name the full hash when declaring it initially.

OBSERVE: Hash Element Population

```
$count{'ants'} = 47_000;
```

This line puts an element in the hash **%count**. The element's key is **'ants'** and its value is **47_000**. (Remember, the underscore in a literal integer is ignored; use it where we would use a comma to write out the entire number.) If there had already been an element in that hash with the **'ants'** key, its value would have been overwritten with **47_000**.

Just as with arrays, each individual value of a hash is a scalar, so when we refer to it, we use a dollar sign **$** to remind us that we're getting a scalar. When you see a dollar sign **$** followed by a name and something inside curly brackets **{}**, you're looking at a reference to a hash value whose key is found within those curly brackets **{}**.

Reading Hash Values

If we use that same syntax inside an expression instead of assigning to it, then we *fetch* the value from the hash corresponding to the key between the curly brackets **{}**. If there is no element in the hash with that key, we get the undefined value. (And yes, you can store the undefined value in a hash value; later we'll find out how to tell the difference between an undefined hash value and a hash element that's not there at all.)

We've used a variable as the key in our example. Now let's modify **emporium.pl** to use a literal as the key:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

my (%count, %cost);
$count{'ants'}        = 47_000;
$count{'fleas'}       = 240_000;
$count{'beetles'}     = 520;
$count{'fruit flies'} = 1_500_000;

$cost{'ants'}        = 0.10;
$cost{'fleas'}       = 0.04;
$cost{'beetles'}     = 0.02;
$cost{'fruit flies'} = 0.001;

print "Value of ants on hand = ", $count{'ants'} * $cost{'ants'}, "\n";
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./emporium.pl
Value of ants on hand = 4700
cold:~/perl1$
```

A hash is like an array, but instead of having a numeric index that starts at zero and increases by one for each element, the hash index is whatever you deem it to be. Instead of square brackets, hash elements have curly brackets and finally, instead of just numbers in the index position, hashes have arbitrary strings.

There is one really helpful piece of syntactic sugar we can use when writing hash elements with literal keys. Change the last line of **emporium.pl** like this:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

my (%count, %cost);
$count{'ants'}        = 47_000;
$count{'fleas'}       = 240_000;
$count{'beetles'}     = 520;
$count{'fruit flies'} = 1_500_000;

$cost{'ants'}        = 0.10;
$cost{'fleas'}       = 0.04;
$cost{'beetles'}     = 0.02;
$cost{'fruit flies'} = 0.001;

print "Value of ants on hand = ", $count{ants} * $cost{ants}, "\n";
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./emporium.pl
Value of ants on hand = 4700
cold:~/perl1$
```

Our changes made no difference. How come?

Well, in Perl it's common to use **well-behaved** strings for literal hash keys (here **literal** means that we spell out the value of the key itself in the program, rather than put it into a variable first), so we don't need to include quotation marks around those strings used as hash keys.

Well-behaved means the same thing here as it did it did when we learned about the **fat comma** (**=>**) operator: if it obeys the rules for naming a scalar *after* the dollar sign (see lesson 2), it works. And by the way, the same rules apply to naming an array or a hash—after the at sign (**@**) or the percent sign (**%**).

There is one key in both hashes in our current example that cannot have the quotation marks left out when using it as a literal key. Can you tell which one it is? (If not, try removing quotation marks until you get an error.)

Hash Initialization

emporium.pl contains hashes that are initialized explicitly; they aren't dependent on data that comes from outside of the program. Most hashes in the real world fall into the latter category; they will be populated by data from an external data source when the program is run.

A proper version of **emporium.pl** would likely read the inventory counts and item prices from data files. But there are still lots of good reasons to use explicitly initialized hashes in a program. For example, suppose we were doing date conversions and we wanted to be able to convert a date with an alphabetical month into one with a numeric month. Until now, you might have coded that like this:

OBSERVE:

```
if ( $month_string eq 'Jan' )
{
  $month_number = 1;
}
elsif ( $month_string eq 'Feb' )
{
  $month_number = 2;
}
...
```

Or you might have come up with an array and a loop, relying on the fact that the corresponding numbers for each month are consecutive small integers that could be array indexes. But now suppose you have a hash **%month_num** with the month strings for keys and the month numbers for values. Your code would slim down drastically, and look something like this:

OBSERVE:

```
$month_number = $month_num{ $month_string };
```

This code is more concise. Instead of varying according to the number of comparisons that have to be made in a multi-way **if** statement or loop, the amount of time required to find hash values from their keys is constant.

We don't need to retrieve **%month_num** from outside of the program because we define it within the program. And so we don't need to set up twelve assignments like we did in **emporium.pl**. We can populate the hash like this:

OBSERVE:

```
my %month_num;
%month_num = ( 'Jan', 1, 'Feb', 2, 'Mar', 3, 'Apr', 4, 'May', 5, 'Jun', 6, 'Jul'
, 7, 'Aug', 8, 'Sep', 9, 'Oct', 10, 'Nov', 11, 'Dec', 12 );
```

A hash is populated by assigning to it a list that will be interpreted as:

(*key, value, key, value, key, value,...*)

There must be an *even* number of elements in the list or else Perl will give you a warning. Each *key* precedes its corresponding *value* in the list.

Even though the list is ordered, when its contents go into the hash, the elements (pairs of values in the list) lose their order. The list *overwrites* any contents that are already in the hash, replacing them with the elements in the list. (If the list is empty, then the hash will be emptied.)

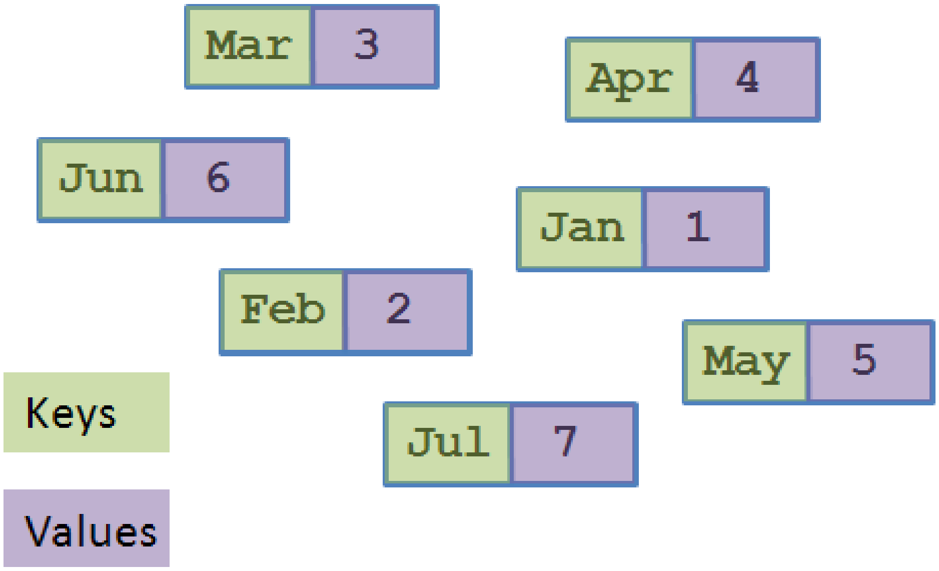
We can combine declaration with population to get initialization. Awesome. But before we go further with that, think back on the syntactic sugar from the last lesson, particularly the "fat comma." Hash population or initialization are ideal places to use it. Here's what initializing **%month_num** looks like with the fat comma:

OBSERVE:

```
my %month_num = ( Jan => 1, Feb => 2, Mar => 3, Apr => 4, May => 5, Jun => 6, Ju
l => 7, Aug => 8, Sep => 9, Oct => 10, Nov => 11, Dec => 12 );
```

Using that fat comma makes it clear what the keys and the values are going to be when the list is assigned to the hash. (And remember that a string on the left side of a fat comma must be well-behaved.)

Visualize your hash like this (not all of the elements have been included in order to keep the diagram from becoming too large):



Iterating through Hashes

The most common action you'll take with a hash is to go through it printing out each key and its corresponding value. You already know how to get the value corresponding to a given key; you read it from **$hash{$key}**. But how do you find a key if you don't know one?

keys()

Enter the **keys()** function. When you pass this function the name of a hash, it returns a list of all the keys in the hash. (Incidentally, although lists have an order, hash keys don't, so the order of the keys returned by **keys()** is unpredictable and not useful.) Modify **emporium.pl** as shown:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

my (%count, %cost);
$count{'ants'}        = 47_000;
$count{'fleas'}       = 240_000;
$count{'beetles'}     = 520;
$count{'fruit flies'} = 1_500_000;

$cost{'ants'}        = 0.10;
$cost{'fleas'}       = 0.04;
$cost{'beetles'}     = 0.02;
$cost{'fruit flies'} = 0.001;

foreach my $insect ( keys %count )
{
  print "Value of $insect on hand = ", $count{$insect} * $cost{$insect}, "\n";
}
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./emporium.pl
Value of ants on hand = 4700
Value of beetles on hand = 10.4
Value of fruit flies on hand = 1500
Value of fleas on hand = 9600
cold:~/perl1$
```

One line is printed for every element in the hash. We called the **keys()** function with the **%count** hash as its argument. Hash elements (and array elements) interpolate in double-quoted strings.

Note This program assumes that every key in **%count** is also a key in **%cost**. It's okay to make that assumption as long as you've checked the program design beforehand to ensure that it will always be valid.

Modify **emporium.pl** as shown:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

my (%count, %cost);
$count{'ants'}        = 47_000;
$count{'fleas'}       = 240_000;
$count{'beetles'}     = 520;
$count{'fruit flies'} = 1_500_000;

$cost{'ants'}        = 0.10;
$cost{'fleas'}       = 0.04;
$cost{'beetles'}     = 0.02;
$cost{'fruit flies'} = 0.001;

foreach my $insect ( keys %count )
{
  print "We have $count{$insect} of $insect on hand at \$$cost{$insect} each\n";
}
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./emporium.pl
We have 47000 of ants on hand at $0.1 each
We have 520 of beetles on hand at $0.02 each
We have 1500000 of fruit flies on hand at $0.001 each
We have 240000 of fleas on hand at $0.04 each
cold:~/perl1$
```

Miscellaneous Hash Functions

Before we learn the many uses for hashes themselves, let's cover a few hash functions:

exists()

When you pass the **exists()** function a hash element, it returns true if that element is in the hash, and false if it isn't. Here's an example:

OBSERVE: exists()

```
if ( exists $cost{spider} )
{
  print "Arachnophobes, beware!\n";
}
```

Hey, wait a minute—isn't this going to *evaluate* the hash element **$cost{spider}** first, and if it's not there, get the undefined value, and then pass that to the **exists()** function? That seems like a violation of Perl's rules, doesn't it?

But it's not. Perl's semantics, like English, aren't entirely regular. Some built-in functions operate—to promote ease of reading—as though they were passed something other than what they are actually passed. This is one of them. Similarly, when you call the **keys()** function, Perl doesn't evaluate the hash in a list context and pass the results of that on to **keys()**. The **keys()** and **exists()** built-in functions are special in this respect; they're doing things that you cannot (yet) write subroutines to do.

The **exists()** function lets us tell the difference between a hash element with an undefined value, and an element that is not present in the hash at all. But most uses for hashes turn out to have values that are not only defined, but true, so you end up testing for the existence of a key with code like this:

OBSERVE: hash element test

```
if ( $cost{spider} )
{
  print "Arachnophobes, beware!\n";
}
```

This test presumes that you'd never stock something with a cost of zero.

delete()

When you pass the **delete()** function a hash element, Perl removes that element from the hash. (If it wasn't in the hash to begin with, Perl doesn't complain.) Take a look:

OBSERVE: delete()

```
foreach my $removed ( @discontinued_items )
{
  delete $cost{$removed};
  delete $count{$removed};
}
```

Once again, Perl is not evaluating the element before passing it to **delete()**, but instead passing the name of the hash and the value of the key as distinct parameters, and using the hash element as a reader-friendly syntax.

delete() returns the value of the element it just removed from the hash. This is often useful for making code succinct.

values()

There is also a **values()** function. The **values()** function is not quite as useful as **keys()** though, because values are not required to be unique and you can't specify a hash element given its value. Also, the order of **values()** is as unpredictable as that of **keys()**, except in this useful situation: if you have just run **keys()** and *have not changed the hash* before running **values()**, then the order of the values will match the order of the keys.

OBSERVE: values()

```
foreach my $value ( values %cost )
{
  print "Selling something for \$$value\n"
}
```

Using Hashes

It's unusual to solve a problem with a Perl program without using a hash somewhere in that code. There are many *patterns* of uses for hashes that show up repeatedly in problem-solving. We'll describe a couple of them here.

Testing Set Membership

Often we want to know whether a given value is a member of some particular set. Is a tomato a fruit or a vegetable? Is a duck-billed platypus a mammal or an amphibian? Is a chickpea a chick or a pea? The hash is ideal for answering these questions because it looks up the answer instantly, instead of scanning through an entire array. It reads all of the members of the set into a hash at the start. (By the end of the course we'll see how to read from a file.) Here's an example of this type of code:

OBSERVE: Set Membership

```
my @tree_names = qw(larch pine oak birch fir palm);
my %is_a_tree;
foreach my $tree (@tree_names)
{
  $is_a_tree{$tree} = 1;
}

my $tree_to_test = 'manzanita';
if ( $is_a_tree{$tree_to_test} )
{
  print "Yup... it's a tree\n";
}
else
{
  print "Not a tree... sure it's not a bush?\n";
}
```

Counting Occurrences

It's common to inquire how many times a particular string (or number, or pattern) shows up in a chunk of text. Maybe you have a web server log file and you want to know how often a particular client hit your server. Or you may have a text version of an e-book and want to know how many times the author used the word "spiffy." Let's find out how many times the word "love" occurs in the Beatles' song, "Love Me Do." Create a file called **word_count.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $text = <<'END_OF_TEXT';
Love, love me do.
You know I love you,
I'll always be true,
So please, love me do.
Whoa, love me do.

Love, love me do.
You know I love you,
I'll always be true,
So please, love me do.
Whoa, love me do.

Someone to love,
Somebody new.
Someone to love,
Someone like you.

Love, love me do.
You know I love you,
I'll always be true,
So please, love me do.
Whoa, love me do.

Love, love me do.
You know I love you,
I'll always be true,
So please, love me do.
Whoa, love me do.
Yeah, love me do.
Whoa, oh, love me do
END_OF_TEXT

my %count;
foreach my $line ( split "\n", $text )
{
  foreach my $word ( split " ", $line )
  {
    $count{$word}++;
  }
}

print "'love' occurs $count{love} times\n";
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./word_count.pl
'love' occurs 18 times
cold:~/perl1$
```

We counted *all* of the words in the lyrics, not just "love." We can postincrement (**++**) an element in the hash if the element doesn't exist yet:

The code above is *syntactic sugar* for:

OBSERVE: postincrement expanded

```
$count{$word} = $count{$word} + 1
```

If Perl evaluates the **$count{$word}** on the right side of the **=** sign, and that element does not yet exist in the hash, the result evaluates as the undefined value (which has the name **undef**). In a *numeric context* (a subset of scalar context, forced in this case by the **+** sign), **undef** is zero, and so the expression becomes adding zero to one to get one, and assigning that to **$count{$word}**. Normally, using **undef** in a numeric context also results in a warning (if we have **use warnings** turned on), but in the *specific* case where the numeric context occurs as a result of using the **++** operator, that warning will *not* appear. (That design choice was made specifically so that this coding idiom would work.)

Let's improve a program from an earlier lesson.

Open **store_report.pl** again in CodeRunner:

OBSERVE: store_report.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $lines = <<'END_OF_REPORT';
 0.95   300   White Peaches
 1.45   120   California Avocados
10.50    10   Durien
 0.40  1500   Spartan Apples
END_OF_REPORT

my @sold = (12, 6, -1, 24);  # One durien returned... too smelly
foreach my $line ( split "\n", $lines )
{
  my ($cost, $quantity, $item) = split " ", $line, 3;
  $quantity -= shift @sold;
  printf "%5.2f %6d %s\n", $cost, $quantity, $item;
}
```

You should be concerned that the array of the number of fruit sold must be ordered the same way as the items in the report. That's *fragile* code; it can break easily. If we were to change a line in the report to a new fruit, nothing would tell us that we should also change the contents of the array **@sold**, so the output would look valid, but it would be wrong.

We need to associate each count of fruit sold with the item it corresponds to; a hash is just the way to do it. Modify **store_report.pl** as shown:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

my $lines = <<'END_OF_REPORT';
 0.95   300   White Peaches
 1.45   120   California Avocados
10.50    10   Durien
 0.40  1500   Spartan Apples
END_OF_REPORT

my %sold = ('White Peaches' => 12, 'Rainier Cherries' => 20,
             Durien          => -1, 'Spartan Apples'   => 24);  # One durien retu
rned... too smelly
foreach my $line ( split "\n", $lines )
{
  my ($cost, $quantity, $item) = split " ", $line, 3;
  if ( exists $sold{$item} )
  {
    $quantity -= $sold{$item};
  }
  else
  {
    warn "Didn't sell any $item this time\n";
  }
  printf "%5.2f %6d %s\n", $cost, $quantity, $item;
}
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./store_report.pl
 0.95    288 White Peaches
Didn't sell any California Avocados this time
 1.45    120 California Avocados
10.50     11 Durien
 0.40   1476 Spartan Apples
cold:~/perl1$
```

Make sure you understand the logic behind these results. You can still use the fat comma after a string that has quotation marks around it, in which case it doesn't have any effect at all, but it's still useful to be able to use the fat comma to differentiate keys from values.

So there you have it—well done! You've just powered through a really important topic that will dramatically improve your Perl coding skills!

And remember: time flies like an arrow, but fruit flies like a banana. Alright then. See you in the next lesson!

Basic Unix Commands and Emacs

Lesson Objectives

When you complete this lesson, you will be able to:

- use basic Unix commands.
- use wild cards to limit the amount of information displayed.
- copy and remove files.
- use a text editor.
- use emacs.

Getting Familiar with Unix

In this lesson you'll learn the basic functions used to write programs in Unix. After all, you won't always create programs in the CodeRunner editor; you'll want to create and manipulate your files in the real world.

Listing Files

Let's get right to it! Click the **New Terminal** button so you see the Unix prompt: **cold:~$**

Your shell contains all the files that you've saved to your OST account. To see a list of those files, use the **ls** command. Most Unix commands have very short names. At first the command names may not make much sense to you, but they do have meaning. In this case, **ls** stands for **l**ist **s**tuff. (At least that's how I remember it! Some people say it's just a shortened version of **l**i**s**t.)

Let's try using the **ls** command. Type at the Unix command line as shown (do NOT change directories this time!):

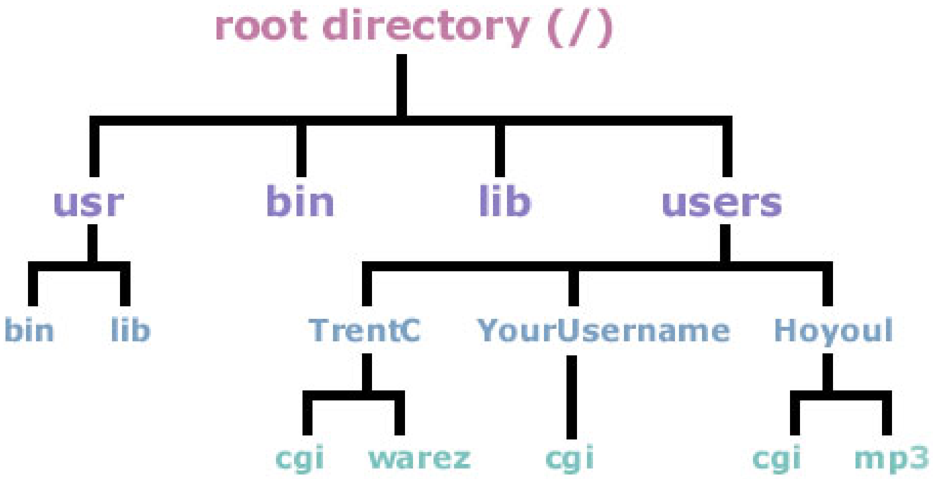
INTERACTIVE TERMINAL SESSION:

```
cold:~$ ls
Objective1.txt          cgi/                index.html              perl1
cold:~$
```

You'll probably see a different list, but even if you've never saved anything to your account, you'll have an index.html file and a cgi directory. These were created automatically when you signed up with OST. If you've had your account for a while (and you have) and taken other courses, you probably have quite a few files stored there.

Some of the "stuff" listed will have a forward slash (/) following the name (for example, **cgi/**). That forward slash tells you it's a directory. Directories provide a way to organize your files. A directory is structured kind of like a tree.

The root directory (indicated by the forward slash /) is the base of the tree. Directories within the root directory are branches, and they in turn may have branches within them, which may have branches within them; you can create subdirectories infinitely, and you can store files in the root directory or in any subdirectory.



To navigate through the tree structure, you'll use the **cd** command. This stands for **c**hange **d**irectory. The **cd** command must be followed by the name of the destination directory.

Try changing to the cgi directory and listing the files there. Type at the Unix command line as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~$ cd cgi
cold:~/cgi$ ls
ajaxlib.pl*  cgi-lib.pl*  counter.pl*  guest.pl*
cold:~/cgi$
```

You got a list that shows all the files in your cgi directory. The command prompt changes to **cold:~/cgi$**. This means you're working in the cgi directory. If you want to get back to your home directory from here, specify **..** as your destination directory. **..** indicates that you want to go back up just one directory.

Now go do some exploring! Keep in mind that you can be transported back to your home directory instantly from any location by typing **cd** with no destination directory.

Wild Cards

If you have hundreds of files in your directory, the **ls** command could return many more files than you care to see. You can use *wild cards* to limit the amount of information displayed.

Make sure you are in the **cgi** directory, then type at the Unix command line as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~/cgi$ ls c*
cgi-lib.pl*  counter.pl*
cold:~/cgi$
```

Only the files with names that begin with the letter **c** are listed. The asterisk (*****) represents the wildcard used to take the place of any number of characters.

Note Keep in mind that Unix is case-sensitive; it can distinguish between upper and lowercase letters.

Directories

Create a new directory within your /perl1 directory and name it **stuff**. You can do that using the **mkdir**, or *make directory*, command. Type at the Unix command line as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~/cgi$ cd ../perl1
cold:~/perl1$ mkdir stuff
cold:~/perl1$
```

Now if you list your files, you can see the new directory. Type at the Unix command line as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ls
(you'll see all the programs we've created, with your new stuff/ folder appearing in alphabetical order in the list.)
```

Now, move into the new directory by using the *change directory* command, **cd**. Type at the Unix command line as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ cd stuff
cold:~/perl1/stuff$
```

List the files in the new directory, using the **-a** flag with the **ls** command. A *flag* is an additional parameter given to a command. The **-a** flag is used to display **a**ll files in a directory. Type at the Unix command line as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1/stuff$ ls -a
./   ../
cold:~/perl1/stuff$
```

What?! How can there already be two files in a new directory? And what's with the names? Weird.

Well, those "files" are actually directories that are part of the Unix file structure. The single dot represents the current directory, and the double dots represent the previous, or "parent" directory. This allows you to use the **cd ..** command to back "up" to that directory. Type at the Unix command line as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1/stuff$  cd ..
cold:~/perl1$
```

Copying and Removing Files

To copy and remove files, use the **cp** and **rm** commands, respectively. Type at the Unix command line as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ cp /etc/issue .
cold:~/perl1$
```

This means "copy the file named **issue** from the **/etc** directory into the current directory (**.**)." Now type at the Unix command line as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ls
issue  stuff/
cold:~/perl1$
```

Again, you'll see the whole list of files you've created in these lessons, and at least the **issue** file and the **stuff/** folder.

Excellent. But you don't really need that file, so let's remove it. Type at the Unix command line as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ rm issue
cold:~/perl1$ ls
stuff/
cold:~/perl1$
```

Using a Text Editor

So far in the course, we've used the CodeRunner editor to create files. In the real world, we have the option of using any of a number different editors—use whichever one you like best. On a Linux-based operating system, you'll usually have access to the Emacs editor.

Command Keys and the Emacs Window

There are various command key sequences in emacs that we use to manipulate, save, search, and exit files. Here are the basic key command sequence elements explained:

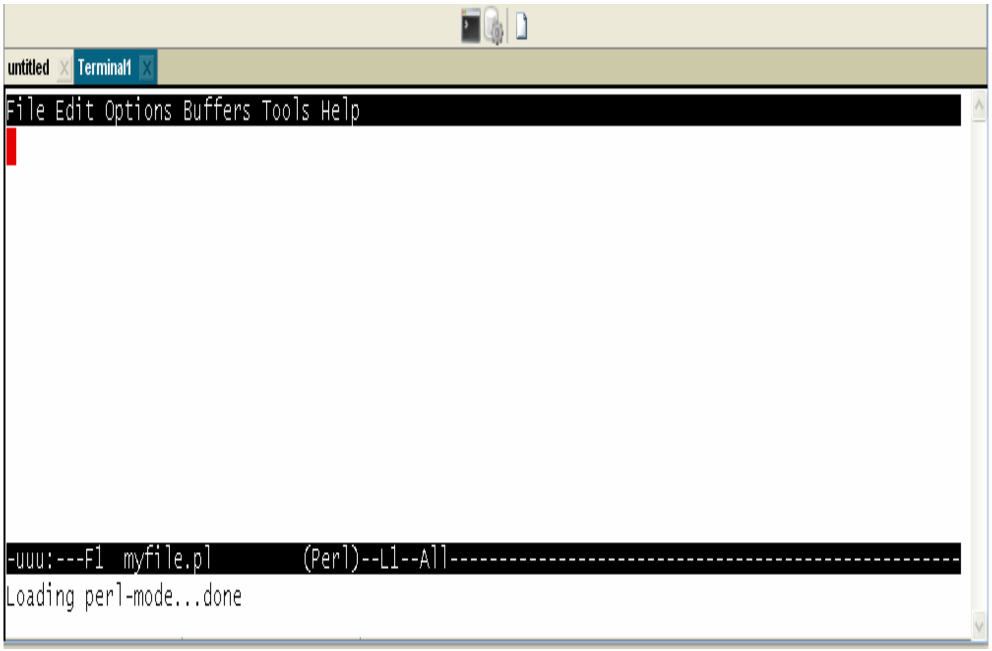
C-g	**C** represents the Ctrl key. **g** represents the G key. The dash (**-**) between the C and G signifies that you should press both keys at the same time.
M-x	**M** represents the META key. But newer computers no longer have a META key, so when you see **M**, press the Esc key instead. Then press the X key. In some instances, you need to press the Esc key *twice* and then press the X key to execute the META function. This may sound confusing now, but you'll get the hang of it.

Emacs is usually accessed from the Unix command line by typing **emacs**, followed by the name of a file. Type at the Unix command line as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ emacs myfile.pl
cold:~/perl1$
```

You'll see a screen that looks something like this:



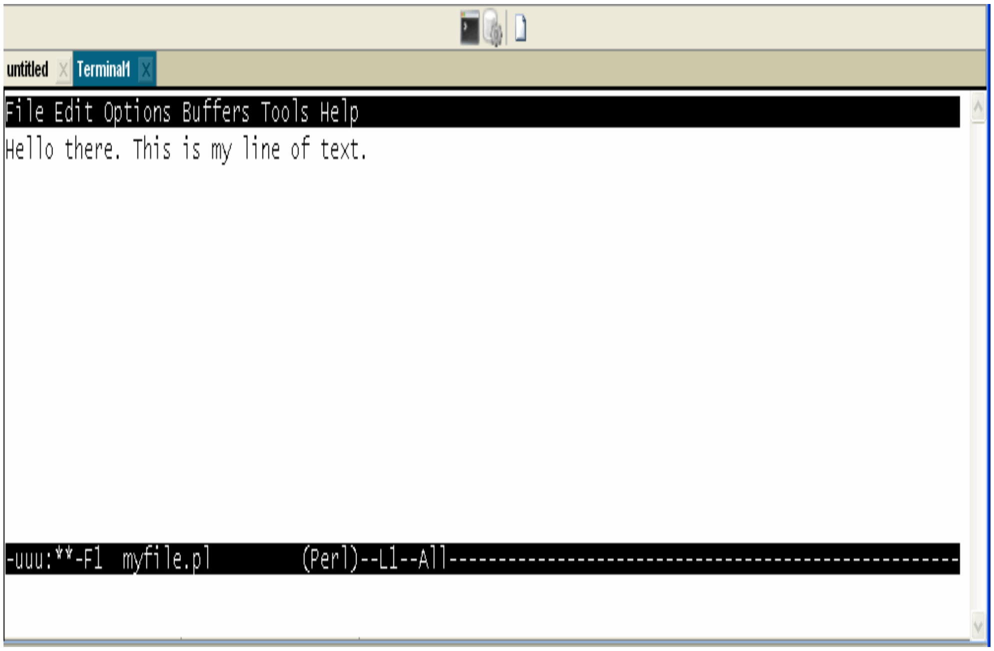
```
-uuu:---F1  myfile.pl        (Perl)--L1--All-----------------------------------------------------
Loading perl-mode...done
```

There's a top heading highlighted in black, then a blank area in the middle, where you'll type. At the bottom there's a line highlighted in black, and then a white line below that.

The black line at the bottom tells you just about everything you'd ever want to know about the current editing environment. You'll see the name of the file currently being edited (**myfile.pl**). Information about the type of file you're editing is located within the parentheses. In this case you are editing a basic text file, so the parentheses contain the word **Text**. The **L1** indicates that your cursor is currently on **line 1** of the document. The last bit of information indicates what portion of the file is on your screen. Right now, it says **All** because you have the entire thing on your screen. Often, it will be a percentage of the document or **Bot** if you happen to be at the end or bottom of your document.

The last line in white is the command line. Whenever you use command keys, they will be displayed there. When you complete a command (such as saving), it will show up there as well.

Type something into the document so you can practice saving and exiting the file. (Emacs will not let you save an empty file):

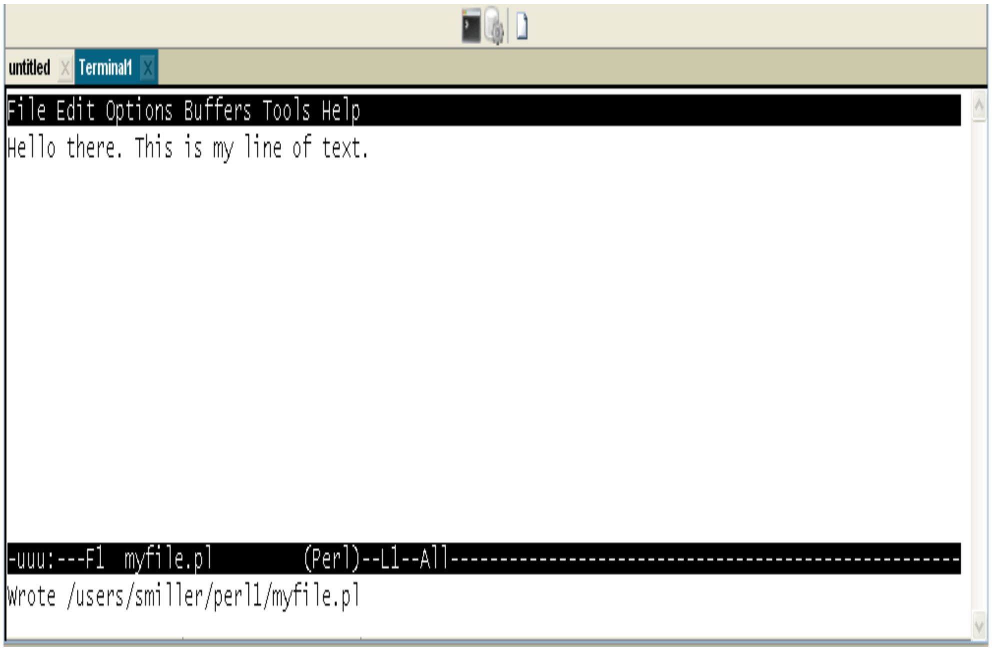


```
Hello there. This is my line of text.

-uuu:**-F1  myfile.pl        (Perl)--L1--All-----------------------------------------------------
```

Note

Because you're using a telnet applet to connect to the server, it does not replicate the emacs program perfectly. After typing text into your document, try using the right and left arrow keys to move back and forth along the text. Depending on the browser and operating system you're using, you may find that some of the characters disappear. If this happens, type **C-l** and your text will reappear. If you have any questions about this, please email your mentor.

Now save **myfile.pl**. Type **C-x** and then **C-s** to save the file. If you make a mistake you can type **C-g** to start over. When you're done, you'll see a message that the file was written in the emacs command line.



```
Hello there. This is my line of text.
```

```
-uuu:---F1  myfile.pl         (Perl)--L1--All-----------------------------------------------------
Wrote /users/smiller/perl1/myfile.pl
```

Exit emacs by typing **C-x C-c**. You'll see a Unix command prompt. Use **ls** to find out if the file is in your account now. Type at the Unix command line as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ls myfile.pl
myfile.pl
cold:~/perl1$
```

Once you're sure that the file has been created, you can run it. Type at the Unix command line:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./myfile.pl
-bash: ./myfile.pl: Permission denied
cold:~/perl1$
```

Hey wait—why doesn't this work? Well, by default, files created in emacs are not created with permission to execute the files. We can fix this though. Type at the Unix command line as shown:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ chmod 755 ./myfile.pl
cold:~/perl1$
```

You're able to run your script now.

Keep in mind that emacs provides many commands for editing files. For more information, check out the built-in **Emacs help** (by pressing **F10** and then **h**) or do a Google search on emacs commands.

In the next lesson we'll learn how to pass arguments to Perl programs at the command line. Good stuff. See you there!

Aggregates, Subroutines, and Command-Line Arguments

Lesson Objectives

When you complete this lesson, you will be able to:

- use aggregates.
- use subroutines.
- write command-line arguments.
- pass arrays to and from subrountines.
- pass hashes to and from subroutines.

"There are no failures—just experiences and your reactions to them."
-Tom Krause

Aggregates, Subroutines, and Command-Line Arguments

Now that we've built a solid foundation, we're ready to refine our understanding of aggregates (arrays and hashes). By the time we finish this lesson, you'll be confident in your ability to use arrays and hashes, and pass them to and from subroutines using the command line.

Command Line Arguments

Retrieving arguments from the command line is pretty straightforward. They arrive into your program in the special array **@ARGV**. A command line argument is anything you type on the command line after the name of your Perl program. If you've made the program executable and refer to it directly with a command such as:

OBSERVE: argumentless command

```
./terraform.pl
```

...then the arguments follow **terraform.pl**, and are separated by spaces:

OBSERVE: command with arguments

```
./terraform.pl -d --jovian "Phase 2 of Invasion Plan" Ganymede Callisto
```

If you invoke the program by calling perl, your command line will look similar:

OBSERVE: command with arguments

```
perl terraform.pl -d --jovian "Phase 2 of Invasion Plan" Ganymede Callisto
```

The rules for forming command line arguments are *not defined by Perl*; they're defined by the command shell you're using, and the program that interprets what you type and subsequently prints the prompt string. When you press **Enter**, that shell interprets the line you typed according to its particular rules for processing, and then determines which information it will give to Perl.

Most programs refer to arguments that begin with hyphens (like **-d** and **--jovian** as we saw in our example) or slashes, as **options** or **flags** and treat them differently. But to the shell, and to Perl, they are regular command line arguments.

White space (spaces or tabs) separates each command line argument and indicates to the shell where one argument ends and another one begins. If you want to enter an argument that contains white space, then you need to surround it with quotation marks, like **"Phase 2 of Invasion Plan"** above. Most shells accept either single or double quotation marks for this purpose.

Alright. Let's give this a try! We'll write a program that does nothing but report on the command line arguments. Create a file called **terraform.pl** as shown:

CODE TO TYPE: terraform.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my $count = 0;
foreach my $argument ( @ARGV )
{
  print '$ARGV[', $count++, '] = ', $argument, "\n";
}
```

Check Syntax and run it without any arguments. Then run it with one argument, and then run it again with three arguments. Try putting quotation marks around an argument containing white space to see how that works. Start a terminal session and type these commands:

INTERACTIVE TERMINAL SESSION

```
cold:~$ cd perl1
cold:~/perl1$ ./terraform.pl
cold:~/perl1$ ./terraform.pl test
$ARGV[0] = test
cold:~/perl1$ ./terraform.pl test test2 test3
$ARGV[0] = test
$ARGV[1] = test2
$ARGV[2] = test3
cold:~/perl1$ ./terraform.pl test "test2 test3"
$ARGV[0] = test
$ARGV[1] = test2 test3
cold:~/perl1$
```

Our command shell parses metacharacters—as most of them do—so try passing some metacharacters in command arguments to see what the shell does with them before Perl sees the arguments. An argument consisting of the asterisk character (*) usually expands to the names of all of the files in the current directory; that's a good metacharacter to try!

The array **@ARGV** is special to Perl, so you don't declare it with **my**—it doesn't belong to you, it belongs to Perl.

One more thing—remember how **shift()** inside a subroutine defaults to shifting **@_**, the array of subroutine arguments? Well, *outside* of subroutines, array **shift()** defaults to **@ARGV**! (By the way, if you're a C programmer and you're wondering what happened to the name of the program, it's in **$0**.)

Passing Arrays To and From Subroutines

Passing arguments to a subroutine is like putting various sticks through a wood chipper—everything that comes out the other end looks the same. Whether you pass scalars, arrays, or hashes, everything gets flattened into a list on the way in and put into an array (**@_**).

An array in a list context expands to the list of the elements in the array. Passing arguments to a subroutine imposes list context.

Here's an example to illustrate these points. Create a file called **interleave.pl** as shown:

CODE TO TYPE: interleave.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my @first = qw(Can unlock secret);
my @second = qw(you the code?);

my @mixed = interleave_words( scalar(@first), @first, @second );
print "Result: @mixed\n";

sub interleave_words
{
  my @results;

  my $count = shift;
  foreach my $index ( 0 .. $count-1 )
  {
    $results[$index * 2] = shift;
  }
  if ( @_ != $count )
  {
    die "Second array not same size ($count) as the first\n";
  }
  foreach my $index ( 0 .. $count-1 )
  {
    $results[$index * 2 + 1] = shift;
  }

  return @results;
}
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./interleave.pl
Result: Can you unlock the secret code?
cold:~/perl1$
```

The subroutine **interleave_words** has been written so that it will interleave the contents of any two arrays that are passed to it. Its first argument is the number of entries in the first array. Why do we do that? After all, doesn't Perl know how many elements there are in an array?

Yes, Perl does know that, but when we pass an array to a subroutine, its identity as an array is lost. Its contents are copied into a list of all the subroutine arguments, which are flattened together into one big long list. (Technically, the contents are not copied but *aliased*, meaning that you could modify them, but as I said before, that's not a good idea, so don't try.)

So, if you pass two arrays to a subroutine, the subroutine can't tell where one array's contents end and the next begin, unless we provide that information, by passing the size of the first array as the very first argument. (This particular subroutine was intended to have two arrays of equal size passed to it. Using a dividing line set halfway through **@_**, it could be written under the assumption that it will always be called correctly, but this leaves no margin for error.)

The **scalar()** function puts its argument into scalar context, so by calling **scalar()** on an array, we are guaranteed to retrieve the size of the array being called.

The result returned from **interleave_words** is an array, which is evaluated in list context because it is being assigned to an array (**@mixed**), and so results in a copy of the array that was **@results** inside the subroutine. Context propagates into subroutines, which is an elegant way of saying that if you want to figure out the context, just imagine that the subroutine call was replaced with whatever follows the word **return** in the subroutine. In this case we have:

OBSERVE: subroutine context propagation

```
my @mixed = interleave_words (...
.
.
.
  return @results;
```

...and so—for the purposes of determining context—imagine that it was saying this instead:

OBSERVE: subroutine context propagation

```
my @mixed = @results;
```

This little trick will save you much time and trouble. If what follows the **return** statement is a call to another subroutine, you can keep going deeper.

Passing Hashes To and From Subroutines

So, what does a hash evaluate as in list context?

It evaluates as a list of key-value pairs, identical to a list that could have been used to initialize the hash to its current state. Each pair consists of two elements in the list, the key followed by the value. The order of the pairs is uncertain, but each value follows its corresponding key in the list.

That allows us to understand how copying a hash works. When you have code that looks like this:

OBSERVE: hash copying

```
my %copy = %original;
```

...the information on the right side of the equals sign (**=**) is placed in list context and evaluates to a list of key-value pairs that is then assigned to the left side of the equals sign (**=**). Previously, you might have looked at that statement and assumed that it resulted in a copy because that's what it looks like, but now you know how it works!

Armed with this knowledge, we can use the subroutine **interleave_words** to perform an entirely different task—creating a hash, given an array of its keys and another array of its values—*without changing* the subroutine definition at all! This is the sort of thing that makes strictly typed languages insanely jealous. Copy **interleave.pl** to **make_hash.pl** and edit it as shown:

CODE TO TYPE: make_hash.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my @keys   = qw(tetra hexa octa dodeca icosa);
my @values = (4, 6, 8, 12, 20);

foreach my $key ( @keys )
{
  $key .= 'hedron';
}
my %faces = interleave_words( scalar(@keys), @keys, @values );
foreach my $solid ( keys %faces )
{
  print "A(n) $solid has $faces{$solid} faces\n";
}

sub interleave_words
{
  my @results;

  my $count = shift;
  foreach my $index ( 0 .. $count-1 )
  {
    $results[$index * 2] = shift;
  }
  if ( @_ != $count )
  {
    die "Second array not same size ($count) as the first\n";
  }
  foreach my $index ( 0 .. $count-1 )
  {
    $results[$index * 2 + 1] = shift;
  }

  return @results;
}
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./make_hash.pl
A(n) tetrahedron has 4 faces
A(n) dodecahedron has 12 faces
A(n) icosahedron has 20 faces
A(n) octahedron has 8 faces
A(n) hexahedron has 6 faces
cold:~/perl1$
```

Hashes do not interpolate in double-quoted strings. We print the contents of this hash with the usual kind of iterative loop.

When I ran this program, it printed the results in this order: 4, 12, 20, 8, 6. What order did your program print them in?

Notice that rather than type the "hedron" suffix many times, I employed a short loop to add it in. In this case, it ended up taking more room than it would have just to type in the longer words, but in a later course you'll find out how to add a suffix like this in just a few words. Right now, we want to get used to the idea of eliminating duplication.

And the **$key** in that loop is an *alias* for each member of the **@keys** array: that is, another name for each element of the array in turn. If it were a *copy* of the element, then the original array elements would not have "hedron" concatenated to them. This aliasing feature is useful for allowing us to modify arrays like we did here, but be careful not to change the loop element of a foreach loop by accident, or you'll overwrite whatever it's aliased to!

Returning a hash from a subroutine isn't too complicated—subroutines return lists, and you assign that list to a hash when you call the subroutine.

Passing Named Arguments to Subroutines

When a subroutine takes more than a couple of arguments, it becomes more challenging to remember the order in which they should be passed. It also becomes more difficult to handle the cases where one or more arguments are optional. Now, Perl doesn't have named parameter passing, but we can pretend that it does by using the fat comma and a hash. Here's how that works:

Step One: when calling the subroutine, precede each argument with a name and a fat comma. For example:

OBSERVE:

```
my $burger_cost = prepare_burger( patties => 2,
                                  condiment => 'mustard',
                                  buns => 'wheat',
                                  tomato => 1 );
```

Step Two: when defining the subroutine, copy the arguments into a hash:

OBSERVE:

```
sub prepare_burger
{
  my %arg = @_;
  # ...
```

Now you can access the arguments as hash entries whose keys are the names of the arguments:

OBSERVE:

```
  # ... continuing the subroutine definition...
  my $cost = 0;
  if ( $arg{tomato} )
  {
    $cost += $tomato_slice_cost;
  }
  $cost += patty_cost * $arg{patties};
  # ...
```

The order in which we pass arguments to the subroutine makes no difference; their names tell us where they are. And, if we want to leave out an optional argument when calling the subroutine, we can tell that from within the subroutine by testing for the existence of the hash element with the argument's name.

To impose rules on the way your subroutine gets called, you can test for compliance by looking at the elements of the hash you copy @_ into. If the rules are not followed, then you can have the subroutine die().

See you in the next lesson!

Sorting

Lesson Objectives

When you complete this lesson, you will be able to:

- incorporate predictability and repeatability into the order of iteration, using a sorted version of the output from **keys()**.
- use Perl's sort function.

"There it was, hidden in alphabetical order."
-Rita Holt

In this lesson, we'll focus on the sorting. Many of the programs we wrote in earlier lessons would have been better if they had incorporated sorting. Take for example, the output of the **keys()** function. Previously, when we iterated through a hash, the output was indeterminate. But we can incorporate predictability and repeatability into the order of iteration, using a sorted version of the output from **keys()**.

sort()

Sorting is a task that helped define computer engineering, back when computer programmers were obsessed with sorting algorithms and code, and computers looked like this:

Computers were so underpowered back then that in order to sort even modest amounts of data, careful attention had to be paid to the method used, to make sure that a task could be completed.

These days, only a few people are still obsessed with improving sorting mechanisms; the rest of us just call the functions that have already been written. The **sort()** function in Perl has been updated over the years to improve its performance, but you don't need to know what goes on inside of it in order to use it. In general, the **sort()** function works like this:

OBSERVE: sort

```
SORTED_LIST = sort UNSORTED_LIST
```

There you are. Not so very complicated, right? Now, let's work on an example. Create a new file called **sort.pl** as shown here:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my @inventory = qw(pears bananas peaches apples cherries oranges lemons grapefruit);
foreach my $fruit ( sort @inventory )
{
  print "$fruit\n";
}
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~$ cd perl1
cold:~/perl1$ ./sort.pl
apples
bananas
cherries
grapefruit
lemons
oranges
peaches
pears
cold:~/perl1$
```

The *input* to the **sort()** function is the array **@inventory**. When the argument list for **sort()** is that straightforward, we usually omit the parentheses—we can omit parentheses around function arguments **if** precedence leads Perl to correctly identify the location of the end of the argument list. It's common practice is to omit parentheses around the arguments to many built-in functions.

The *output* from the **sort()** function is the list that results from sorting **@inventory**. Rather than store that output in another array, I'm using it as the list in a **foreach** statement.

Let's modify that output, starting with the addition of some uppercase letters:

CODE TO TYPE: sort.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my @inventory = qw(pears bananas Peaches apples Cherries oranges lemons Grapefruit);
foreach my $fruit ( sort @inventory )
{
  print "$fruit\n";
}
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./sort.pl
Cherries
Grapefruit
Peaches
apples
bananas
lemons
oranges
pears
cold:~/perl1$
```

The sorting order in Perl is called "ASCIIbetical", meaning that it is not quite alphabetical—as you see by the output you just generated—but instead, it's determined by the *ASCII collating sequence*. ASCII is the **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange, and it's the basis for how characters are stored in computers. (ASCII has been replaced in large part by Unicode, but the transition is still a work in progress and character encoding issues beyond 7-bit ASCII are outside the scope of this course.)

The output in our example is in this particular order, because capital letters come before lowercase letters in ASCII. Here are 7-bit printable characters ordered left-to-right by their position in the ASCII collating sequence:

!"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~

Lots of *non*printable characters are part of that sequence as well. You can find out more by typing "man ascii" at the command prompt on a Unix/Linux computer, or by Googling "ASCII."

In a moment we'll go over the way to sort that list without regard to the case of the letters. But first, let's look at a few functions. Create a new file called **sort_numeric.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my @numbers;
foreach my $x ( 1 .. 10 )
{
  push @numbers, int(rand 1000);
}
foreach my $number ( sort @numbers )
{
  print "$number\n";
}
```

Check Syntax ⚙ and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./sort_numeric.pl
405
497
513
518
677
722
723
842
90
981
cold:~/perl1$
```

We have some new functions in this example:

- **rand**(***N***), which produces a random floating point number greater than or equal to 0, and less than *N* (that is, the result may be exactly 0, but will never be exactly *N*. If you omit *N*, it uses **1** by default).
- **int()**, which returns the integer part of its argument (truncates towards zero).

So **int**(**rand(1000)**) returns a random integer between 0 and 999.

Each time you run that program, you should get different output. Run it as many times as you need to until the output includes a number that isn't three digits long. Where is it in the sorted output? Is that where you think it should be?

The number is where it *should* be—Perl doesn't make mistakes—but you may have had different expectations. That number comes out there because of ASCIIbetical sorting, which treats every element as a string. That means that "123" sorts before "35" just the same as "abc" sorts before "de." If you want to sort numerically, then you have to tell Perl to treat the elements of the **sort** input list as numbers, not strings.

We do that by adding a special argument to **sort()**. It's special because **sort()** already takes a list as its input argument, so there's no room for another argument—if another argument was added, it would look like part of the list to be sorted. There's a special syntax for this special argument; it looks like this:

OBSERVE:

```
SORTED_LIST = sort { $a <=> $b } UNSORTED_LIST
```

That's right, the argument appears within curly brackets between **sort** and the input list. And what's inside those brackets must be *exactly* what's written above. Modify **sort_numeric.pl** as shown:

CODE TO EDIT:

```
#!/usr/bin/perl
use strict;
use warnings;

my @numbers;
foreach my $x ( 1 .. 10 )
{
  push @numbers, int(rand 1000);
}
foreach my $number ( sort { $a <=> $b } @numbers )
{
  print "$number\n";
}
```

Check Syntax and run it. Your output will look something like this:

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./sort_numeric.pl
17
168
201
222
512
670
680
695
716
777
cold:~/perl1$
```

So that's how you sort numbers. (Want to break it? You know you do! Try putting something non-numeric in the input list to **sort()** when it has that special argument and see what happens.)

Now, this bit of information might satisfy a lesser scholar, but not you! You want to *understand* what's going on in detail. So let's dig a little deeper into that special argument.

Between those curly brackets is what's called a *comparison function*. In general, it looks like this:

OBSERVE:

```
SORTED_LIST = sort { COMPARISON } UNSORTED_LIST
```

When Perl sorts a list, deep down inside the clever algorithm that does the sorting, it compares a pair of elements in the list, many times over, for different pairs of elements. Perl wants to determine whether those two elements are equal, and if not, whether the first one is "less than" or "greater than" the other. Armed with that information, the clever algorithm decides how to order those two elements in the sorted output list it is building.

The **<=>** operator (affectionately known as the "spaceship operator"—think "Star Wars") is the basic building block of the numeric comparison function. It returns -1 if the argument on its left is less than the argument on its right, 0 if the arguments are equal, or 1 if the argument on the left is greater than the argument on the right.

There is a *default* comparison function. If you don't supply one—if you just say **sort(@list)**—then it is as if you had written this:

OBSERVE: default comparison

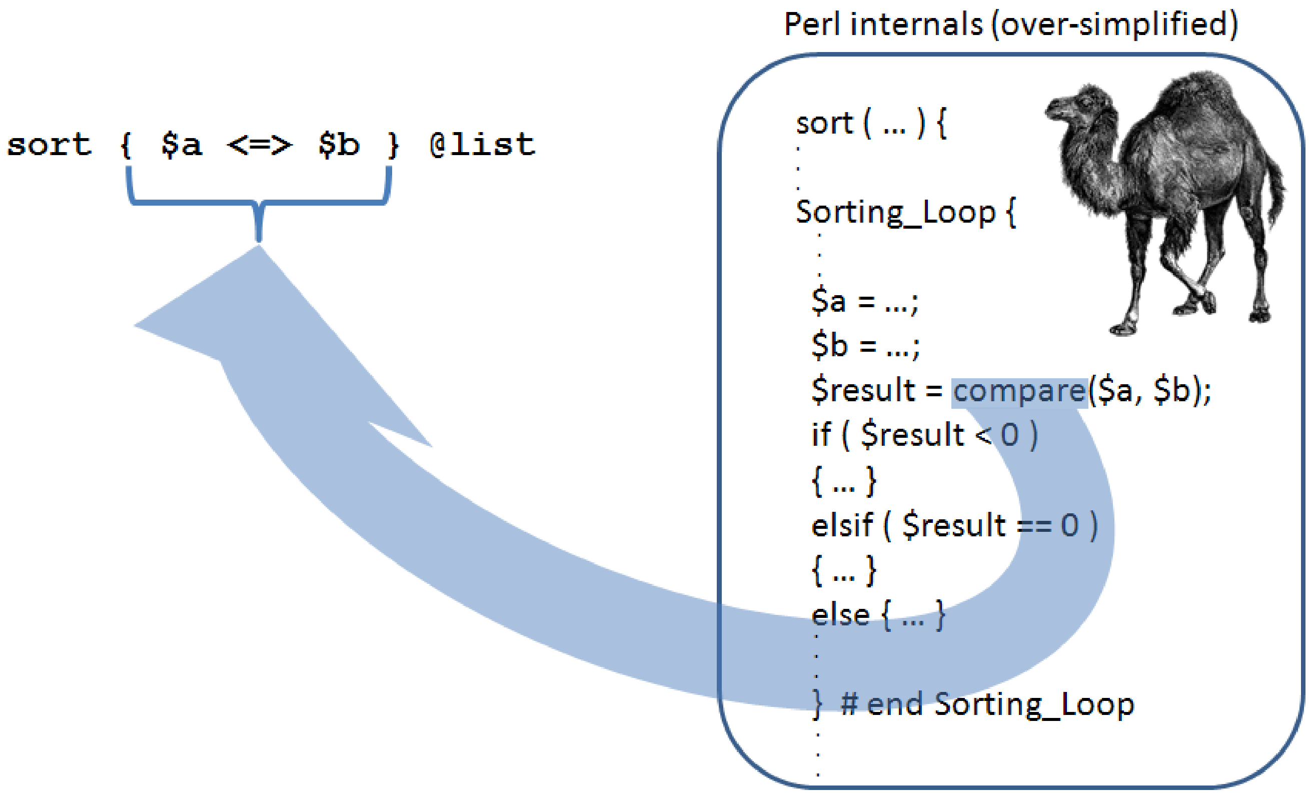
```
sort { $a cmp $b } UNSORTED_LIST
```

That **cmp** operator is to strings what the **<=>** operator is to numbers: it returns -1 if the string on its left is less than (like using the **lt** operator) the string on its right, 0 if the strings are equal, or 1 if the string on its left is greater than (like using the **gt** operator) the string on its right). By the way, you'll almost never see **cmp** or **<=>** used outside of sorting comparison functions.

Now, what are **$a** and **$b**?

They're the two elements from the input list need to be compared by Perl's internal sorting algorithm! (We engineers call this comparison function a *callback*.) When Perl is ready to compare two elements in the list, it sets **$a** and **$b** to those elements, and calls your comparison function (the code between the curly brackets). Perl reads the result, and depending on whether the answer is less than, equal to, or greater than, zero, it determines the way that pair of elements should be ordered.

This diagram shows you what's going on:



```
sort { $a <=> $b } @list
```

Perl internals (over-simplified)

```
sort ( ... ) {
.
.
Sorting_Loop {
 .
 .
 $a = ...;
 $b = ...;
 $result = compare($a, $b);
 if ( $result < 0 )
 { ... }
 elsif ( $result == 0 )
 { ... }
 else { ... }
 .
 .
 } # end Sorting_Loop
 .
 .
```

Custom Sorting Orders

If ASCIIbetical and numeric were the only two ways of sorting lists, we wouldn't have bothered to learn about the comparison function. But there is an *infinite* number of possible comparison functions! You can write them to sort any way you want. Let's say, for example, that you want to sort strings without regard to their case; in other words, make "apples" sort before "Peaches". Go ahead and modify the code in **sort.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my @inventory = qw(pears bananas Peaches apples Cherries oranges lemons Grapefruit);
@inventory = sort { lc($a) cmp lc($b) } @inventory;
foreach my $fruit ( @inventory )
{
  print "$fruit\n";
}
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./sort.pl
apples
bananas
Cherries
Grapefruit
lemons
oranges
Peaches
pears
cold:~/perl1$
```

The comparison function puts every argument that it's passed into lowercase; so when it is passed "Peaches" and "bananas", it will return the result of **"peaches" cmp "bananas"** (note that "peaches" is now in lowercase). Now Perl's sort function knows that "bananas" goes before "Peaches" when sorted. Give it a try!

I deliberately sorted **@inventory** in place to show you that it could be done; there are no issues with putting the results into the same array as the original input is located.

More advanced sorting functions can figure out how to arrange dates typed in this format "29/08/2009" into something that can be sorted:

OBSERVE: Date Sorting

```
sort { date_to_num( $a ) <=> date_to_num( $b ) } ...
sub date_to_num
{
  my $date  = shift;

  my $day   = substr $date, 0, 2;
  my $month = substr $date, 3, 2;
  my $year  = substr $date, 6, 4;

  return "$year$month$day";
}
```

Even though we concatenated strings together into another string to be returned from **date_to_str**, Perl was still able to sort the result numerically because the strings looked like numbers. However, since the strings are all of the same length, string sorting would have worked equally well (until the year 10,000).

(You may remember **substr()** from the homework for lesson 10!)

Every comparison function ends up constructing either two numbers compared with **<=>** or two strings compared with **cmp**.

Integrated Example

Let's add sorting to one of our long-running examples. Modify **store_report.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my $lines = <<'END_OF_REPORT';
 0.95   300   White Peaches
 1.45   120   California Avocados
10.50    10   Durien
 0.40  1500   Spartan Apples
 1.50   400   Cherry Tomatoes
END_OF_REPORT

my (%item_cost, %inventory);
foreach my $line ( split "\n", $lines )
{
  my ($cost, $quantity, $item) = split " ", $line, 3;
  $item_cost{$item} = $cost;
  $inventory{$item} = $quantity;
}

my %sold = ('White Peaches' => 12, 'Rainier Cherries' => 20,
             Durien          => -1, 'Spartan Apples'   => 24);  # One durien returned...
 too smelly

foreach my $item ( keys %sold )
{
  if ( exists $inventory{$item} )
  {
    $inventory{$item} -= $sold{$item};
  }
  else
  {
    warn "*** Sold $sold{$item} of $item, which were not in inventory\n";
  }
}

foreach my $item ( sort keys %item_cost )
{
  printf "%5.2f %6d %s\n", $item_cost{$item}, $inventory{$item}, $item;
}
```

Check Syntax and run it.

INTERACTIVE TERMINAL SESSION:

```
cold:~/perl1$ ./store_report.pl
*** Sold 20 of Rainier Cherries, which were not in inventory
1.45    120 California Avocados
1.50    400 Cherry Tomatoes
10.50     11 Durien
0.40   1476 Spartan Apples
0.95    288 White Peaches
cold:~/perl1$
```

Take a gander at these features:

- The inventory is now captured internally in two hashes—**%item_cost** and **%inventory**—before processing any transactions. This is a common programming practice: read the external data into internal data structures before manipulating them. (I know, the data in this case isn't really "external," since it comes from strings inside the program, but we're going to learn how to get data from outside the program in the next lesson!)
- Don't bother sorting the keys of **%sold** when processing the sales. It wouldn't make any difference,

because there is no output from this phase.

- *Do* sort the key of **%item_cost**, because output is being produced, and it's good practice to produce that output in a predictable order.
- We didn't apply case-insensitive sorting because we assume that the input data will be consistent with respect to the use of capital letters. Feel free to change this!
- In this program **%inventory** and **%item_cost** must contain the same keys. The way that it is structured, this is guaranteed.

Can you believe it? We're getting ready to take on our final lesson. See you in there!

Reading External Files

Lesson Objectives

When you complete this lesson, you will be able to:

- input woth Perl's <> operator.
- read external files.
- use he built-in chomp function.

Welcome to the final lesson in your Introduction to Perl course! This last lesson involves learning how to read information from files outside of your program.

Input with Perl's <> Operator

Command line programs, in particular the "filtering" set of programs that are in the Unix/Linux utility set (**grep, sed, awk, head, tail, cut, sort** and, yes, even **cat**) all have a common "interface." You invoke each of these programs using a command line call that looks like this:

program options file(s)

In our command line, ***program*** would be a member of our filtering set, such as **awk** or **sed**. Then ***options*** (if there are options present) would consist of a set of options, each beginning with a minus sign (**-**). Then ***file(s)*** would be one or more filenames, which would then be read in succession by ***program***, and processed.

If no files are specified, the ***program*** reads its input from standard input (*program* is usually called as part of a command pipeline set up with the vertical bar (**|**) symbol between programs). Standard input may also be read at a specific point if a filename argument is the minus sign by itself. Take a look at a few samples to familiarize yourself with this style of program invocation:

- cat amphibians.txt rodents.txt > mammals.txt
- head -100 wind_in_the_willows | sed -e 's/toad/hedgehog/g' > plagiarized_story
- tail -f error_log | grep -vi php
- generate_orders -month 10 | sort -nr orders_sep - orders_nov > orders_fall

Here's a skeleton for a program in Perl using those same calling conventions:

OBSERVE:

```
#!/usr/bin/perl
use strict;
use warnings;

# Shift off any arguments in @ARGV that begin with minus signs, and any values they tak
e
while ( defined( my $line = <> ) )
{
  # Process a line of input, which is now in $line
}
```

This skeleton provides *all* of the functionality we talked about, *including* processing multiple files, and processing standard input. And all that functionality is implemented by just two characters: **<>**. (They are called the "diamond operator" because of the shape formed by putting the characters together.)

We'd better take a closer look at those two characters! Even though at first glance you might think so, they don't have anything to do with less than or greater than tests. In fact, this pair of characters together comprise a special operator in Perl—and that operator does a *lot*. We'll begin to see just how much, in this example. Create **store_report.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my (%item_cost, %inventory);
while ( defined( my $line = <> ) )
{
  my ($cost, $quantity, $item) = split " ", $line, 3;
  $item_cost{$item} = $cost;
  $inventory{$item} = $quantity;
}

my %sold = ('White Peaches' => 12, 'Rainier Cherries' => 20,
             Durien          => -1, 'Spartan Apples'   => 24);  # One durien returned...
 too smelly

foreach my $item ( keys %sold )
{
  if ( exists $inventory{$item} )
  {
    $inventory{$item} -= $sold{$item};
  }
  else
  {
    warn "*** Sold $sold{$item} of $item, which were not in inventory\n";
  }
}

foreach my $item ( sort keys %item_cost )
{
  printf "%5.2f %6d %s\n", $item_cost{$item}, $inventory{$item}, $item;
}
```

Note Keep in mind that this program is not the same version you used in your homework that accepted sales figures from the command line; we're going to use that command line here for passing the store inventory filename.

Now, create an input file named **store_report.input** in the same folder:

TEXT TO TYPE:

```
 0.95   300   White Peaches
 1.45   120   California Avocados
10.50    10   Durien
 0.40  1500   Spartan Apples
 1.50   400   Cherry Tomatoes
```

Save both of your files, and then Check Syntax and run store_report.pl.

INTERACTIVE TERMINAL SESSION:

```
cold:~$ cd perl1
cold:~/perl1$ ./store_report.pl ./store_report.input
*** Sold 24 of Spartan Apples, which were not in inventory
*** Sold -1 of Durien, which were not in inventory
*** Sold 20 of Rainier Cherries, which were not in inventory
*** Sold 12 of White Peaches, which were not in inventory
1.45    120 California Avocados

1.50    400 Cherry Tomatoes

10.50     10 Durien

0.40   1500 Spartan Apples

0.95    300 White Peaches

cold:~/perl1$
```

It *doesn't work*. (It runs, but doesn't produce the same output as the last version of this program.) Can you see why? What's odd about the inventory that it prints out?

The inventory output has a blank line between each line of output. What does that suggest? *That each item in the inventory—the last thing printed on each line—contains a newline character at the end of it.* That explains why none of the sold items were found in the inventory: the sales loop was looking in **%inventory** for a key of, for instance, **"Durien"**, but that key wasn't there. Instead, there was an element with a key of **"Durien\n"** in **%inventory**.

chomp()

When **<>** reads in a line, it leaves the **\n** at the end of the line. The previous version of this program used **split "\n"**, **$lines**. The result didn't contain any **\n** characters, because the **split** function returned everything *between* the **\n** characters.

The built-in function **chomp()** takes one argument, a scalar, and if a **\n** character is present at the end of the scalar, that **\n** character is removed. If the scalar doesn't end with a **\n**, then **chomp()** has no effect. Here's how you'll use **chomp()**:

OBSERVE:

```
chomp $line
```

Go ahead and modify **store_report.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my (%item_cost, %inventory);
while ( defined( my $line = <> ) )
{
  chomp $line;
  my ($cost, $quantity, $item) = split " ", $line, 3;
  $item_cost{$item} = $cost;
  $inventory{$item} = $quantity;
}

my %sold = ('White Peaches' => 12, 'Rainier Cherries' => 20,
             Durien          => -1, 'Spartan Apples'   => 24);  # One durien retu
rned... too smelly

foreach my $item ( keys %sold )
{
  if ( exists $inventory{$item} )
  {
    $inventory{$item} -= $sold{$item};
  }
  else
  {
    warn "*** Sold $sold{$item} of $item, which were not in inventory\n";
  }
}

foreach my $item ( sort keys %item_cost )
{
  printf "%5.2f %6d %s\n", $item_cost{$item}, $inventory{$item}, $item;
}
```

Check Syntax and run it.

INTERACTIVE SESSION:

```
cold:~/perl1$ ./store_report.pl ./store_report.input
*** Sold 20 of Rainier Cherries, which were not in inventory
1.45    120 California Avocados
1.50    400 Cherry Tomatoes
10.50     11 Durien
0.40   1476 Spartan Apples
0.95    288 White Peaches
cold:~/perl1$
```

Your results are the same as when you ran a version of this program in the previous lesson.

Now let's see the "default to standard input" behavior of **<>**. Run the same program again *without any arguments.* Nothing will happen. Perl is waiting for input on STDIN, which is by default set to your terminal. Copy and paste some of the lines from **store_report.input** (or copy them from the display below, or make up your own new input). Then type **Ctrl-D** *at the beginning of the next line*. That is how you tell the Unix shell that there is no more input on STDIN, and that causes Perl to see an end of file on input. You will not get your prompt back until you type **Ctrl-D** on a line by itself! I've shown it as **^D** below:

INTERACTIVE SESSION:

```
cold:~/perl1$ ./store_report.pl
 0.95   300   White Peaches
 1.45   120   California Avocados
10.50    10   Durien
^D
*** Sold 20 of Rainier Cherries, which were not in inventory
*** Sold 24 of Spartan Apples, which were not in inventory
1.45    120 California Avocados
10.50     11 Durien
0.95    288 White Peaches
cold:~/perl1$
```

You don't have to supply input to STDIN by typing it like this, though. You can use the redirection operators of the Unix shell to supply it from another program, for instance.

Now that you have a working example of **<>**, we'll go over its behavior in detail.

Operation of <> in Scalar Context

When you call **<>** for the *first* time, it examines **@ARGV**.

If there are no elements in **@ARGV**, then **<>** operates on standard input.

If there is an element in **@ARGV**, Perl removes that element from **@ARGV** (Perl actually modifies the **@ARGV** array from within the **<>** operator).

Then Perl opens the file named by that element. (If the file cannot be opened, Perl prints a warning to the standard error stream and goes on to the next element of **@ARGV**.)

If the file name is the minus sign (**-**), Perl opens standard input.

Now that Perl has an input file open, it returns the first line of that file. That's the (scalar context) result of **<>**.

The *next* time that **<>** is called, it returns the *next* line from the input, and so on.

When there are no more lines in the input, **<>** returns the undefined value.

Operation of <> in List Context

In list context, **<>** returns all remaining lines from all remaining files named in **@ARGV**.

When **<>** has finished (returns **undef** in scalar context or has finished executing in list context), **@ARGV** will be empty.

Going Further with the Example

Now let's alter **store_report.pl** so it will read the sales report from a file as well. Modify **@ARGV** *yourself* so it contains just the filename(s) you need at each point. (Even though **@ARGV** is a special variable set by Perl, we're still allowed to change it!) Edit **store_report.pl** as shown:

CODE TO TYPE:

```
#!/usr/bin/perl
use strict;
use warnings;

my %sold;
if ( @ARGV > 2 && shift eq "-s" )
{
  my $sales_file = shift;
  my @saved_argv = @ARGV;
  @ARGV = $sales_file;
  while ( defined( my $line = <> ) )
  {
    chomp $line;
    my ($quantity, $item) = split " ", $line, 2;
    $sold{$item} = $quantity;
  }
  @ARGV = @saved_argv;
}
else
{
  die "Usage: $0 -s sales_file inventory file...\n";
}

my (%item_cost, %inventory);
while ( defined( my $line = <> ) )
{
  chomp $line;
  my ($cost, $quantity, $item) = split " ", $line, 3;
  $item_cost{$item} = $cost;
  $inventory{$item} = $quantity;
}

~~my %sold = ('White Peaches' => 12, 'Rainier Cherries' => 20,~~
~~              Durien          => -1, 'Spartan Apples'   => 24);   # One durien returned...~~
~~ too smelly~~

foreach my $item ( keys %sold )
{
  if ( exists $inventory{$item} )
  {
    $inventory{$item} -= $sold{$item};
  }
  else
  {
    warn "*** Sold $sold{$item} of $item, which were not in inventory\n";
  }
}

foreach my $item ( sort keys %item_cost )
{
  printf "%5.2f %6d %s\n", $item_cost{$item}, $inventory{$item}, $item;
}
```

Now create **sales_report.sales** with the following contents:

TEXT TO TYPE:

```
12 White Peaches
20 Rainier Cherries
-1 Durien
24 Spartan Apples
```

The program name is followed by an option, **-s**, which takes an argument that is the name of a file from sales data. (Ironically, this option is not optional.) Any inventory files you want to include must be named after the sales data file on the command line.

Check Syntax and run store_report.pl as shown:

INTERACTIVE SESSION:

```
cold:~/perl1$ ./store_report.pl -s ./sales_report.sales ./store_report.input
*** Sold 20 of Rainier Cherries, which were not in inventory
1.45     120 California Avocados
1.50     400 Cherry Tomatoes
10.50     11 Durien
0.40    1476 Spartan Apples
0.95     288 White Peaches
cold:~/perl1$
```

Your program's output should be the same as the output you got the last time you ran it.

First, our program makes sure that **@ARGV** contains enough elements and that the first element is **"-s"**. Next, it **shift**s off the name of the sales file, saves the remainder of **@ARGV**, and repopulates **@ARGV** with just the name of the sales file. Then it reads the sales file using the diamond operator, which reads only the sales file because that's the only file named in **@ARGV**. Then our program restores the saved contents of **@ARGV** and goes on with the rest of the program as usual.

In the usage message, the special variable **$0** is set by Perl to contain the name of the program itself. Try calling the program with an incorrect usage to trigger the error. Now try altering the items and quantities in the sales and inventory files, to see what happens.

I should point out that this is not the most common skeleton for processing files named on the command line in Perl. The more common method is outside the scope of this course, but it's covered in Perl 2, along with lots of other cool stuff!

We're all done here!

Congratulations, you did it! We're at the end of the course. Thanks for making this happen and for all of your hard work! It's been great having you here. Remember to finish your final assignments, and always have fun using Perl!