ECOL 553L
Perl Basics: Variables, Arrays and Hashes
• We've seen the she-bang line that specifies the path to the Perl interpreter:
• We've seen the she-bang line that specifies the path to the Perl interpreter:
  • `#!/usr/bin/perl`
We've seen the she-bang line that specifies the path to the Perl interpreter:
  • #!/usr/bin/perl

Other lines that begin with # are Comments
Perl Comment Lines

• We've seen the she-bang line that specifies the path to the Perl interpreter:
  • `#!/usr/bin/perl`

• Other lines that begin with `#` are Comments
• Comments can also start within a line:
Perl Comment Lines

- We've seen the she-bang line that specifies the path to the Perl interpreter:
  - `#!/usr/bin/perl`

- Other lines that begin with `#` are Comments
- Comments can also start within a line:
  - `$str = “yada” x 3;  # repeat yada 3 times`
Perl Comment Lines

• We've seen the she-bang line that specifies the path to the Perl interpreter:
  • `#!/usr/bin/perl`

• Other lines that begin with # are Comments
• Comments can also start within a line:
  • `$str = "yada" x 3;  # repeat yada 3 times`

• Use comments to document the purpose of your script and to explain segments of code
• We've seen the she-bang line that specifies the path to the Perl interpreter:
  • `#!/usr/bin/perl`

• Other lines that begin with # are Comments
• Comments can also start within a line:
  • `$str = "yada" x 3;  # repeat yada 3 times`

• Use comments to document the purpose of your script and to explain segments of code
  • `# Remember to add a newline (\n) character to output`
Perl Comment Lines

• We've seen the she-bang line that specifies the path to the Perl interpreter:
  • `#!/usr/bin/perl`

• Other lines that begin with # are Comments
• Comments can also start within a line:
  • `$str = “yada” x 3;  # repeat yada 3 times`

• Use comments to document the purpose of your script and to explain segments of code
  • # Remember to add a newline (\n) character to output
  • `print "If you are called 'Sam', some files share your name! \n";`
Perl Comment Lines

• We've seen the she-bang line that specifies the path to the Perl interpreter:
  • `#!/usr/bin/perl`

• Other lines that begin with `#` are Comments
• Comments can also start within a line:
  • `$str = "yada" x 3;  # repeat yada 3 times`

• Use comments to document the purpose of your script and to explain segments of code
  • `# Remember to add a newline (\n) character to output`
  • `print "If you are called 'Sam', some files share your name! \n";`

• If you include good comments in your code, you'll thank yourself when you return to the code in six months or a year or more…
Perl Basic Data Types and Scalar variables

- Numeric data: integers, floating point, scientific
Perl Basic Data Types and Scalar variables

- Numeric data: integers, floating point, scientific
  - Examples: 13, 2.7182818, 2e-40, 1.6e200
Perl Basic Data Types and Scalar variables

- Numeric data: integers, floating point, scientific
  - Examples: 13, 2.7182818, 2e-40, 1.6e200
- String data:
Perl Basic Data Types and Scalar variables

- Numeric data: integers, floating point, scientific
  - Examples: 13, 2.7182818, 2e-40, 1.6e200
- String data:
  - Examples: "Perez Hilton", "13", "#10 Downing St."
Perl Basic Data Types and Scalar variables

• Numeric data: integers, floating point, scientific
  • Examples: 13, 2.7182818, 2e-40, 1.6e200

• String data:
  • Examples: "Perez Hilton", "13", "#10 Downing St."

• Variables store pieces of data or information inside a Perl script
Perl Basic Data Types and Scalar variables

• Numeric data: integers, floating point, scientific
  • Examples: 13, 2.7182818, 2e-40, 1.6e200
• String data:
  • Examples: "Perez Hilton", "13", "#10 Downing St."

• Variables store pieces of data or information inside a Perl script
• A scalar variable can hold one single piece of data, which can be a string or a numeric type:
Perl Basic Data Types and Scalar variables

• Numeric data: integers, floating point, scientific
  • Examples: 13, 2.7182818, 2e-40, 1.6e200

• String data:
  • Examples: "Perez Hilton", "13", "#10 Downing St."

• Variables store pieces of data or information inside a Perl script
• A scalar variable can hold one single piece of data, which can be a string or a numeric type:
  • $name = "Perez Hilton";
Perl Basic Data Types and Scalar variables

- Numeric data: integers, floating point, scientific
  - Examples: 13, 2.7182818, 2e-40, 1.6e200
- String data:
  - Examples: "Perez Hilton", "13", "#10 Downing St."

Variables store pieces of data or information inside a Perl script
- A scalar variable can hold one single piece of data, which can be a string or a numeric type:
  - $name = "Perez Hilton";
  - $lucky_num = 13;
Perl Basic Data Types and Scalar variables

• Numeric data: integers, floating point, scientific
  • Examples: 13, 2.7182818, 2e-40, 1.6e200

• String data:
  • Examples: "Perez Hilton", "13", "#10 Downing St."

• Variables store pieces of data or information inside a Perl script
• A scalar variable can hold one single piece of data, which can be a string or a numeric type:
  • $name = "Perez Hilton";
  • $lucky_num = 13;
  • $letter = "C";
Perl Basic Data Types and Scalar variables

- Numeric data: integers, floating point, scientific
  - Examples: 13, 2.7182818, 2e-40, 1.6e200

- String data:
  - Examples: "Perez Hilton", "13", "#10 Downing St."

- Variables store pieces of data or information inside a Perl script
- A scalar variable can hold one single piece of data, which can be a string or a numeric type:
  - $name = "Perez Hilton";
  - $lucky_num = 13;
  - $letter = "C";
  - $e_value = 1.3e-40;
Perl Basic Data Types and Scalar variables

- Numeric data: integers, floating point, scientific
  - Examples: 13, 2.7182818, 2e-40, 1.6e200
- String data:
  - Examples: "Perez Hilton", "13", "#10 Downing St."

- Variables store pieces of data or information inside a Perl script
- A scalar variable can hold one single piece of data, which can be a string or a numeric type:
  - $name = "Perez Hilton";
  - $lucky_num = 13;
  - $letter = "C";
  - $e_value = 1.3e-40;
- Scalar variables have names that begin with $
Perl Operators

- Arithmetic operators: +  -  *  /  **
Perl Operators

- Arithmetic operators: +  -  *  /  **
  - $sum = 6 + 7;
Perl Operators

- Arithmetic operators: + - * / **
  - $sum = 6 + 7;
  - $sum = $sum + 1;
Perl Operators

- Arithmetic operators: +  -  *  /   **
  - $sum = 6 + 7;
  - $sum = $sum + 1;
  - $avg = $sum / $count;
Perl Operators

- Arithmetic operators: +  -  *  /  **
  - $sum = 6 + 7;
  - $sum = $sum + 1;
  - $avg = $sum / $count;
  - $max16bit = 2 ** 16 - 1;
Perl Operators

- Arithmetic operators: +  -  *  /   **
  - $sum = 6 + 7;
  - $sum = $sum + 1;
  - $avg = $sum / $count;
  - $max16bit = 2 ** 16 - 1;

- String operators:  .   x
Perl Operators

• Arithmetic operators: +  -  *  /  **
  • $sum = 6 + 7;
  • $sum = $sum + 1;
  • $avg = $sum / $count;
  • $max16bit = 2 ** 16 - 1;

• String operators:  .  x
  • $first = "Sam";
Perl Operators

- Arithmetic operators: + - * / **
  - $sum = 6 + 7;
  - $sum = $sum + 1;
  - $avg = $sum / $count;
  - $max16bit = 2 ** 16 - 1;

- String operators: . x
  - $first = "Sam";
  - $last = "Rockwell";
Perl Operators

- **Arithmetic operators:** +  -  *  /  **
  - $sum = 6 + 7;
  - $sum = $sum + 1;
  - $avg = $sum / $count;
  - $max16bit = 2 ** 16 - 1;

- **String operators:** .  x
  - $first = "Sam";
  - $last = "Rockwell";
  - $full = $first . " " . $last;
Perl Operators

- **Arithmetic operators:** +  -  *  /  **
  - $sum = 6 + 7;
  - $sum = $sum + 1;
  - $avg = $sum / $count;
  - $max16bit = 2 ** 16 - 1;

- **String operators:** .  x
  - $first = "Sam";
  - $last = "Rockwell";
  - $full = $first . " " . $last;
  - $three_cheers = "Rah" x 3;
Perl Operators

- **Arithmetic operators:** +  -  *  /  **
  - $sum = 6 + 7;
  - $sum = $sum + 1;
  - $avg = $sum / $count;
  - $max16bit = 2 ** 16 - 1;

- **String operators:** .  x
  - $first = "Sam";
  - $last = "Rockwell";
  - $full = $first . " " . $last;
  - $three_cheers = "Rah" x 3;

- **Comparison operators:** <  <=  ==  !=  >=  >  eq  ne
Perl Operators

- **Arithmetic operators:** $sum = 6 + 7;  
  $sum = $sum + 1;  
  $avg = $sum / $count;  
  $max16bit = 2 ** 16 - 1;

- **String operators:** $first = "Sam";  
  $last = "Rockwell";  
  $full = $first . " " . $last;  
  $three_cheers = "Rah" x 3;

- **Comparison operators:** if ($sum <= 100) { ... }

```perl
$sum = 6 + 7;
$sum = $sum + 1;
$avg = $sum / $count;
$max16bit = 2 ** 16 - 1;

$first = "Sam";
$last = "Rockwell";
$full = $first . " " . $last;
$three_cheers = "Rah" x 3;
```
Perl Operators

- Arithmetic operators: +  -  *  /  **
  - $sum = 6 + 7;
  - $sum = $sum + 1;
  - $avg = $sum / $count;
  - $max16bit = 2 ** 16 - 1;

- String operators:  .  x
  - $first = "Sam";
  - $last = "Rockwell";
  - $full = $first . " " . $last;
  - $three_cheers = "Rah" x 3;

- Comparison operators: <  <=  ==  !=  >=  >  eq  ne
  - if ($sum <= 100) { ... }
  - if ($first ne "Sam") { ... }
Perl Operators

• Arithmetic operators: + - * / **
  • $sum = 6 + 7;
  • $sum = $sum + 1;
  • $avg = $sum / $count;
  • $max16bit = 2 ** 16 - 1;

• String operators: . x
  • $first = "Sam";
  • $last = "Rockwell";
  • $full = $first . " " . $last;
  • $three_cheers = "Rah" x 3;

• Comparison operators: < <= == != >= > eq ne
  • if ($sum <= 100) { ... }
  • if ($first ne "Sam") { ... }

• Logical operators: && || !
Perl Operators

- **Arithmetic operators:** +  −  *  /  **
  - $sum = 6 + 7;
  - $sum = $sum + 1;
  - $avg = $sum / $count;
  - $max16bit = 2 ** 16 − 1;

- **String operators:** .  x
  - $first = "Sam";
  - $last = "Rockwell";
  - $full = $first . " " . $last;
  - $three_cheers = "Rah" x 3;

- **Comparison operators:** <  <=  ==  !=  >=  >  eq  ne
  - if ($sum  <=  100) { … }
  - if ($first  ne  "Sam") { … }

- **Logical operators:** &&  ||  !
  - if ($first  ne  "Sam" &&  $sum  <=  100) { … }
Perl Operators

- **Arithmetic operators:** + - * / **
  - $sum = 6 + 7;
  - $sum = $sum + 1;
  - $avg = $sum / $count;
  - $max16bit = 2 ** 16 - 1;

- **String operators:** . x
  - $first = "Sam";
  - $last = "Rockwell";
  - $full = $first . " " . $last;
  - $three_cheers = "Rah" x 3;

- **Comparison operators:** < <= == != >= > eq ne
  - if ($sum <= 100) { ... }
  - if ($first ne "Sam") { ... }

- **Logical operators:** && || !
  - if ($first ne "Sam" && $sum <= 100) { ... }
  - if (! $found) { # keep looking! ... }
More about Perl Operators
We've seen that Perl operators have a precedence hierarchy, e.g. multiplication has a higher precedence than addition. You can use parentheses to alter order of evaluation:
We've seen that Perl operators have a precedence hierarchy, e.g. multiplication has a higher precedence than addition. You can use parentheses to alter order of evaluation:

```
$value = (7 + 6) * 3;
```
More about Perl Operators

• We've seen that Perl operators have a precedence hierarchy, e.g. multiplication has a higher precedence than addition. You can use parentheses to alter order of evaluation:
  • $value = (7 + 6) * 3;

• With logical operations, "short circuit" evaluation is used. For example, in the if statement below, $sum never gets tested… why?
More about Perl Operators

• We've seen that Perl operators have a precedence hierarchy, e.g. multiplication has a higher precedence than addition. You can use parentheses to alter order of evaluation:
  • $value = (7 + 6) * 3;

• With logical operations, "short circuit" evaluation is used. For example, in the if statement below, $sum never gets tested… why?
  • $first = "Beverly";
More about Perl Operators

• We've seen that Perl operators have a precedence hierarchy, e.g. multiplication has a higher precedence than addition. You can use parentheses to alter order of evaluation:
  • $value = (7 + 6) * 3;

• With logical operations, "short circuit" evaluation is used. For example, in the if statement below, $sum never gets tested… why?
  • $first = "Beverly";
  • $sum = 86 + 50;
• We've seen that Perl operators have a precedence hierarchy, e.g. multiplication has a higher precedence than addition. You can use parentheses to alter order of evaluation:
  • $value = (7 + 6) * 3;

• With logical operations, "short circuit" evaluation is used. For example, in the if statement below, $sum never gets tested… why?
  • $first = "Beverly";
  • $sum = 86 + 50;
  • if ($first eq "Sam" && $sum <= 100)
    { ... }
Using variables with print
Using variables with print

- Variables can be used inside double quoted strings. This is called variable interpolation:
Using variables with print

- Variables can be used inside double quoted strings. This is called variable interpolation:
  - $name = "Barney";
Variables can be used inside double quoted strings. This is called variable interpolation:

- `$name = "Barney";`
- `$age = 90;`
Using variables with print

- Variables can be used inside double quoted strings. This is called variable interpolation:
  - $name = "Barney";
  - $age = 90;
  - print "$name is $age years old...get the wheelchair! \n";
Using variables with print

- Variables can be used inside double quoted strings. This is called variable interpolation:
  - $name = "Barney";
  - $age = 90;
  - print "$name is $age years old...get the wheelchair! \n";
- An example in "Beginning Perl":
Using variables with print

- Variables can be used inside double quoted strings. This is called variable interpolation:
  - $name = "Barney";
  - $age = 90;
  - print "$name is $age years old...get the wheelchair! \n";

- An example in "Beginning Perl":
  - $name = "fred";
• Variables can be used inside double quoted strings. This is called variable interpolation:
  • $name = "Barney";
  • $age = 90;
  • print "$name is $age years old...get the wheelchair! \n";

• An example in "Beginning Perl":
  • $name = "fred";
  • print "My name is ", $name, "\n";
Using variables with print

• Variables can be used inside double quoted strings. This is called variable interpolation:
  • $name = "Barney";
  • $age = 90;
  • print "$name is $age years old...get the wheelchair! \n";

• An example in "Beginning Perl":
  • $name = "fred";
  • print "My name is ", $name, "\n";

• The same thing can be done this way:
Using variables with print

• Variables can be used inside double quoted strings. This is called variable interpolation:
  • $name = "Barney";
  • $age = 90;
  • print "$name is $age years old...get the wheelchair! \n";

• An example in "Beginning Perl":
  • $name = "fred";
  • print "My name is ", $name, "\n";

• The same thing can be done this way:
  • $name = "fred";
Using variables with print

• Variables can be used inside double quoted strings. This is called variable interpolation:
  • $name = "Barney";
  • $age = 90;
  • print "$name is $age years old...get the wheelchair! \n";

• An example in "Beginning Perl":
  • $name = "fred";
  • print "My name is ", $name, "\n";

• The same thing can be done this way:
  • $name = "fred";
  • print "My name is $name \n";
• Variables can be used inside double quoted strings. This is called variable interpolation:
  • $name = "Barney";
  • $age = 90;
  • print "$name is $age years old...get the wheelchair! 
";

• An example in "Beginning Perl":
  • $name = "fred";
  • print "My name is ", $name, "\n";

• The same thing can be done this way:
  • $name = "fred";
  • print "My name is $name \n";

• Or this way:
• Variables can be used inside double quoted strings. This is called variable interpolation:

$\$name = "Barney";
$\$age = 90;
print "$name is $age years old...get the wheelchair! \n";

• An example in "Beginning Perl":

$\$name = "fred";
print "My name is ", $name, "\n";

• The same thing can be done this way:

$\$name = "fred";
print "My name is $name \n";

• Or this way:

$\$name = "fred";
Using variables with print

- Variables can be used inside double quoted strings. This is called variable interpolation:
  - $name = "Barney";
  - $age = 90;
  - print "$name is $age years old…get the wheelchair! \n";

- An example in "Beginning Perl":
  - $name = "fred";
  - print "My name is ", $name, "\n";

- The same thing can be done this way:
  - $name = "fred";
  - print "My name is $name \n";

- Or this way:
  - $name = "fred";
  - print "My name is " . $name . "\n";
my $name = <STDIN>;
print "name is $name \n";
{
    my $name = "Lucy";
    print "\t inside the block, name is $name \n";
}
print "outside the block, name is $name \n";
Reading input from the keyboard

- Scripts are much more useful if they can read input from the user. The construct for doing this is:

```perl
my $name = <STDIN>;
print "name is $name \n";
{
  my $name = "Lucy";
  print "\t inside the block, name is $name \n";
}
print "outside the block, name is $name \n";
```
Scripts are much more useful if they can read input from the user. The construct for doing this is:

```perl
my $var = <STDIN>;
```

```perl
my $name = <STDIN>;
print "name is $name \n";
{
    my $name = "Lucy";
    print "\t inside the block, name is $name \n";
}
print "outside the block, name is $name \n";
```
Reading input from the keyboard

- Scripts are much more useful if they can read input from the user. The construct for doing this is:
  - `$var = <STDIN>;`

- `STDIN` is an example of a "handle" and we will learn more about these later. The `<>` surrounding `STDIN` is called the "diamond" operator, and it reads one line of input.

```perl
my $name = <STDIN>;
print "name is $name \n";
{
    my $name = "Lucy";
    print "\t inside the block, name is $name \n";
}
print "outside the block, name is $name \n";
```
Scripts are much more useful if they can read input from the user. The construct for doing this is:

- $var = <STDIN>;

STDIN is an example of a "handle" and we will learn more about these later. The <> surrounding STDIN is called the "diamond" operator, and it reads one line of input.

We will practice with this script:

```perl
my $name = <STDIN>;
print "name is $name \n";
{
    my $name = "Lucy";
    print "\t inside the block, name is $name \n";
}
print "outside the block, name is $name \n";
```
my $name = "fred";  # Here $name has File Scope
print "name is $name \n";
{
    my $name = "lucy"; # Here $name has block scope
    print "inside the block, name is $name \n";
}
print "out here, name is $name \n";
A Bit About Variable Scope

- The scope of a variable refers to its lifetime within a script, i.e. when and where it is accessible.

my $name = "fred";  # Here $name has File Scope
print "name is $name \n";
{
    my $name = "lucy";  # Here $name has block scope
    print "inside the block, name is $name \n";
}
print "out here, name is $name \n";
A Bit About Variable Scope

• The scope of a variable refers to its lifetime within a script, i.e. when and where it is accessible.
• Sometimes you will want to restrict the scope of a variable using the my keyword, so that it won't interfere or collide with another variable having the same name. This will be very important later when we learn about subroutines.

my $name = "fred";  # Here $name has File Scope
print "name is $name \n";
{
  my $name = "lucy";  # Here $name has block scope
  print "inside the block, name is $name \n";
}
print "out here, name is $name \n";
A Bit About Variable Scope

- The scope of a variable refers to its lifetime within a script, i.e. when and where it is accessible.
- Sometimes you will want to restrict the scope of a variable using the `my` keyword, so that it won't interfere or collide with another variable having the same name. This will be very important later when we learn about subroutines.
- Here is an example:

```perl
my $name = "fred";  # Here $name has File Scope
print "name is $name \n";
{
    my $name = "lucy"; # Here $name has block scope
    print "inside the block, name is $name \n";
}
print "out here, name is $name \n";
```
Short cuts with Operators
Short cuts with Operators

• We can add to a scalar variable like this:
Short cuts with Operators

- We can add to a scalar variable like this:
  - `$sum = $sum + 8;`
Short cuts with Operators

- We can add to a scalar variable like this:
  - `$sum = $sum + 8;`
- Or we can use shorthand that combines the operator and =
Short cuts with Operators

• We can add to a scalar variable like this:
  • $\text{sum} = \text{sum} + 8;$

• Or we can use shorthand that combines the operator and =
  • $\text{sum} += 8; \quad # \text{add 8 to sum}$
Short cuts with Operators

- We can add to a scalar variable like this:
  - $sum = $sum + 8;
- Or we can use shorthand that combines the operator and =
  - $sum += 8;  # add 8 to sum
- This works for most operators:
Short cuts with Operators

- We can add to a scalar variable like this:
  - `$sum = $sum + 8;`
- Or we can use shorthand that combines the operator and =
  - `$sum += 8;  # add 8 to sum`
- This works for most operators:
  - `$product *= 12;  # multiply product by 12`
Short cuts with Operators

- We can add to a scalar variable like this:
  - `$sum = $sum + 8;`
- Or we can use shorthand that combines the operator and =
  - `$sum += 8;   # add 8 to sum`
- This works for most operators:
  - `$product *= 12;   # multiply product by 12`
- Adding and subtracting one are so commonly done that there are even shorter shortcuts!
Short cuts with Operators

- We can add to a scalar variable like this:
  - $\textit{sum} = \textit{sum} + 8$
- Or we can use shorthand that combines the operator and =
  - $\textit{sum} += 8; \# \text{ add 8 to sum}$
- This works for most operators:
  - $\textit{product} *= 12; \# \text{ multiply product by 12}$

- Adding and subtracting one are so commonly done that there are even shorter shortcuts!
  - $\textit{year} = \textit{year} + 1; \# \text{ add 1 to year}
Short cuts with Operators

• We can add to a scalar variable like this:
  • \$sum = \$sum + 8;

• Or we can use shorthand that combines the operator and =
  • \$sum += 8;  # add 8 to sum

• This works for most operators:
  • \$product *= 12;  # multiply product by 12

• Adding and subtracting one are so commonly done that there are even shorter shortcuts!
  • \$year = \$year + 1;  # add 1 to year
  • \$year += 1;       # add 1 to year
Short cuts with Operators

- We can add to a scalar variable like this:
  - `$sum = $sum + 8;`
- Or we can use shorthand that combines the operator and `=`
  - `$sum += 8;  # add 8 to sum`
- This works for most operators:
  - `$product  *=  12;  # multiply product by 12`
- Adding and subtracting one are so commonly done that there are even shorter shortcuts!
  - `$year  =  $year + 1;  # add 1 to year`
  - `$year  +=  1;  # add 1 to year`
  - `$year++;  # add 1 to year`
Short cuts with Operators

• We can add to a scalar variable like this:
  • $sum = $sum + 8;

• Or we can use shorthand that combines the operator and =
  • $sum += 8;   # add 8 to sum

• This works for most operators:
  • $product  *=  12;   # multiply product by 12

• Adding and subtracting one are so commonly done that there are even shorter shortcuts!
  • $year  =  $year + 1;   # add 1 to year
  • $year  +=  1;          # add 1 to year
  • $year++;               # add 1 to year
  • $total--;              # subtract 1 from total
Short cuts with Operators

• We can add to a scalar variable like this:
  • $sum = $sum + 8;

• Or we can use shorthand that combines the operator and =
  • $sum += 8;    # add 8 to sum

• This works for most operators:
  • $product *= 12;    # multiply product by 12

• Adding and subtracting one are so commonly done that there are even shorter shortcuts!
  • $year = $year + 1;    # add 1 to year
  • $year += 1;          # add 1 to year
  • $year++;               # add 1 to year

  • $total--;              # subtract 1 from total

• These are called autoincrement ++ and autodecrement --
Common Mistakes to watch out for!
Common Mistakes to watch out for!

- Forgetting the `;` at the end of a statement
Common Mistakes to watch out for!

• Forgetting the ; at the end of a statement

• Mismatched parentheses ( ), braces { }, brackets [ ], or quotes ′ ′, " ", ` `
Common Mistakes to watch out for!

- Forgetting the `;` at the end of a statement
- Mismatched parentheses `( )`, braces `{ }`, brackets `[ ]`, or quotes `' '`, " "`, \ ` ``
- Forgetting the `$` at the beginning of a variable name (bare word error)
Common Mistakes to watch out for!

- Forgetting the `;` at the end of a statement
- Mismatched parentheses `()`, braces `{ }`, brackets `[]`, or quotes `' '`, " "`, ` `.
- Forgetting the `$` at the beginning of a variable name (bare word error)
- A mistake on the first line of the script (improper specification of the Perl interpreter – or possibly a space where it shouldn’t be)
Common Mistakes to watch out for!

- Forgetting the `;` at the end of a statement
- Mismatched parentheses `()` , braces `{ }` , brackets `[ ]` , or quotes `' ' , " " , ` ` 
- Forgetting the `$` at the beginning of a variable name (bare word error)
- A mistake on the first line of the script (improper specification of the Perl interpreter – or possibly a space where it shouldn’t be)
- Forgetting to close an output file (can cause missing output)
Common Mistakes to watch out for!

• Forgetting the ; at the end of a statement
• Mismatched parentheses ( ), braces { }, brackets [ ], or quotes ’ ’, " "
• Forgetting the $ at the beginning of a variable name (bare word error)
• A mistake on the first line of the script (improper specification of the Perl interpreter – or possibly a space where it shouldn’t be)
• Forgetting to close an output file (can cause missing output)
• Saving your script to a different folder than the folder you are running it from!
Common Mistakes to watch out for!

- Forgetting the `;` at the end of a statement
- Mismatched parentheses `( )`, braces `{ }`, brackets `[ ]`, or quotes ‘ ’, " ", ` `.
- Forgetting the `$` at the beginning of a variable name (bare word error)
- A mistake on the first line of the script (improper specification of the Perl interpreter – or possibly a space where it shouldn’t be)
- Forgetting to close an output file (can cause missing output)
- Saving your script to a different folder than the folder you are running it from!
- For more ideas, see "Beginning Perl", chapter 9
Today's Topics:

• Introduction to Perl, Part 3
Today's Topics:
Today's Topics:

• Good Perl coding practices
Today's Topics:

- Good Perl coding practices
- use warnings; use strict;
Today's Topics:

- Good Perl coding practices
  - use warnings; use strict;
- Perl Lists
• Today's Topics:
  • Good Perl coding practices
    • use warnings; use strict;
  • Perl Lists
  • Perl Arrays
Today's Topics:

- Good Perl coding practices
  - use warnings; use strict;
- Perl Lists
- Perl Arrays
- Perl Array Functions
Today's Topics:

- Good Perl coding practices
  - use warnings; use strict;
- Perl Lists
- Perl Arrays
- Perl Array Functions
  - push, pop, shift, unshift
Introduction to Perl, Part 3

Today's Topics:

- Good Perl coding practices
  - use warnings; use strict;
- Perl Lists
- Perl Arrays
- Perl Array Functions
  - push, pop, shift, unshift
- Stepping (iterating) through Perl Arrays
Today's Topics:

- Good Perl coding practices
  - use warnings; use strict;
- Perl Lists
- Perl Arrays
- Perl Array Functions
  - push, pop, shift, unshift
- Stepping (iterating) through Perl Arrays
  - foreach, $_
Good Practice: use strict;
• To write better Perl code, include use warnings; and use strict;
Good Practice: use strict;

- To write better Perl code, include use warnings; and use strict;
  - See "Beginning Perl" chapter 9
Good Practice: use strict;

• To write better Perl code, include use warnings; and use strict;
  • See "Beginning Perl" chapter 9
• These tools are called pragmas, and they help to minimize "unsafe" code such as possible variable name collisions and misspellings.
Good Practice: use strict;

- To write better Perl code, include `use warnings;` and `use strict;`
  - See "Beginning Perl" chapter 9
- These tools are called pragmas, and they help to minimize "unsafe" code such as possible variable name collisions and misspellings.
- The warnings pragma tells the Perl interpreter to output warnings when it sees possible typos (common syntax errors such as missing punctuation)
Good Practice: use strict;

• To write better Perl code, include use warnings; and use strict;
  • See "Beginning Perl" chapter 9
• These tools are called pragmas, and they help to minimize "unsafe" code such as possible variable name collisions and misspellings.
• The warnings pragma tells the Perl interpreter to output warnings when it sees possible typos (common syntax errors such as missing punctuation)
• The strict pragma requires all variables to be explicitly declared with keywords such as my or our prior to use.
Good Practice: use strict;

• To write better Perl code, include `use warnings;` and `use strict;`
  • See "Beginning Perl" chapter 9
• These tools are called pragmas, and they help to minimize "unsafe" code such as possible variable name collisions and misspellings.
• The warnings pragma tells the Perl interpreter to output warnings when it sees possible typos (common syntax errors such as missing punctuation)
• The strict pragma requires all variables to be explicitly declared with keywords such as my or our prior to use.
• To write the safer code for web-based scripts, look into using taint mode to protect against malicious user input:
Good Practice: use strict;

• To write better Perl code, include use warnings; and use strict;
  • See "Beginning Perl" chapter 9
• These tools are called pragmas, and they help to minimize "unsafe" code such as possible variable name collisions and misspellings.
• The warnings pragma tells the Perl interpreter to output warnings when it sees possible typos (common syntax errors such as missing punctuation)
• The strict pragma requires all variables to be explicitly declared with keywords such as my or our prior to use.
• To write the safer code for web-based scripts, look into using taint mode to protect against malicious user input:
  • http://www.webreference.com/programming/perl/taint/
Perl Lists

- Recall that a scalar variable can hold one single piece of data:
Perl Lists

• Recall that a scalar variable can hold one single piece of data:
  • assign values to the $name and $lucky_num scalar variables
Perl Lists

• Recall that a scalar variable can hold one single piece of data:
  • assign values to the $name and $lucky_num scalar variables
  • $name = "Greg Bear";
Perl Lists

• Recall that a scalar variable can hold one single piece of data:
  • assign values to the $name and $lucky_num scalar variables
  • $name = "Greg Bear";
  • $lucky_num = 42;
Recall that a scalar variable can hold one single piece of data:
- assign values to the $name and $lucky_num scalar variables
- $name = "Greg Bear";
- $lucky_num = 42;

Scalar variables have names that begin with $
Perl Lists

• Recall that a scalar variable can hold one single piece of data:
  • assign values to the $name and $lucky_num scalar variables
  • $name = "Greg Bear";
  • $lucky_num = 42;
• Scalar variables have names that begin with $
• We can use Perl's list notation to assign values to more than one variable at a time:
Recall that a scalar variable can hold one single piece of data:

- assign values to the $name and $lucky_num scalar variables
  - $name = "Greg Bear";
  - $lucky_num = 42;

Scalar variables have names that begin with $.

We can use Perl's list notation to assign values to more than one variable at a time:

- the following line does the same thing as the two code lines above
• Recall that a scalar variable can hold one single piece of data:
  • assign values to the $name and $lucky_num scalar variables
    • $name = "Greg Bear";
    • $lucky_num = 42;

• Scalar variables have names that begin with $

• We can use Perl's list notation to assign values to more than one variable at a time:
  • the following line does the same thing as the two code lines above
    • ($name, $lucky_num) = ("Greg Bear", 42);
Perl Lists

• Recall that a scalar variable can hold one single piece of data:
  • assign values to the $name and $lucky_num scalar variables
  • $name = "Greg Bear";
  • $lucky_num = 42;
• Scalar variables have names that begin with $
• We can use Perl's list notation to assign values to more than one variable at a time:
  • the following line does the same thing as the two code lines above
  • ($name, $lucky_num) = ("Greg Bear", 42);
• Perl also has a range .. notation:
Recall that a scalar variable can hold one single piece of data:
  assign values to the $name and $lucky_num scalar variables
  $name = "Greg Bear";
  $lucky_num = 42;
Scalar variables have names that begin with $
We can use Perl's list notation to assign values to more than one variable at a time:
  the following line does the same thing as the two code lines above
  ($name, $lucky_num) = ("Greg Bear", 42);
Perl also has a range .. notation:
  print "Range from -30 to 5: ", (-30 .. 5), "\n";
Perl Lists

• Recall that a scalar variable can hold one single piece of data:
  • assign values to the $name and $lucky_num scalar variables
    • $name = "Greg Bear";
    • $lucky_num = 42;

• Scalar variables have names that begin with $

• We can use Perl's list notation to assign values to more than one variable at a time:
  • the following line does the same thing as the two code lines above
    • ($name, $lucky_num) = ("Greg Bear", 42);

• Perl also has a range .. notation:
  • print "Range from -30 to 5: ", (-30 .. 5), "\n";
  • print "Range from J to V: ", ('J' .. 'V'), "\n";
Perl Arrays

- We can store a list of values in an array variable:
Perl Arrays

- We can store a list of values in an array variable:
  - assign values to the @items array variable
Perl Arrays

• We can store a list of values in an array variable:
  • assign values to the @items array variable
  • @items = ("Greg Bear", 42, "X", 3.5e-107);
• We can store a list of values in an array variable:
  • assign values to the @items array variable
  • @items = ("Greg Bear", 42, "X", 3.5e-107);

• Notice that array variable names begin with @
Perl Arrays

• We can store a list of values in an array variable:
  • assign values to the @items array variable
  • @items = ("Greg Bear", 42, "X", 3.5e-107);

• Notice that array variable names begin with @
• To access a single element in an array we use a scalar name and a number called the array index inside square brackets [ ]. Array indices begin with zero!
Perl Arrays

- We can store a list of values in an array variable:
  - assign values to the @items array variable
  - @items = ("Greg Bear", 42, "X", 3.5e-107);

- Notice that array variable names begin with @
- To access a single element in an array we use a scalar name and a number called the array index inside square brackets [ ]. Array indices begin with zero!
  - print the name of an author
Perl Arrays

• We can store a list of values in an array variable:
  • assign values to the @items array variable
    • @items = ("Greg Bear", 42, "X", 3.5e-107);

• Notice that array variable names begin with @

• To access a single element in an array we use a scalar name and a number called the array index inside square brackets [ ]. Array indices begin with zero!
  • print the name of an author
    • print "Darwin's Radio was written by ", $items[0], "\n";
Perl Arrays

- We can store a list of values in an array variable:
  - assign values to the @items array variable
  - @items = ("Greg Bear", 42, "X", 3.5e-107);

- Notice that array variable names begin with @
- To access a single element in an array we use a scalar name and a number called the array index inside square brackets [ ]. Array indices begin with zero!
  - print the name of an author
  - print "Darwin's Radio was written by ", $items[0], 

- print the answer to the universe
Perl Arrays

• We can store a list of values in an array variable:
  • assign values to the @items array variable
    • @items = ("Greg Bear", 42, "X", 3.5e-107);

• Notice that array variable names begin with @
• To access a single element in an array we use a scalar name and a number
called the array index inside square brackets [ ]. Array indices begin with zero!
  • print the name of an author
    • print "Darwin's Radio was written by ", $items[0], "\n";

  • print the answer to the universe
    • print $items[1], "\n";
Perl Arrays

• We can store a list of values in an array variable:
  • assign values to the @items array variable
  • @items = ("Greg Bear", 42, "X", 3.5e-107);

• Notice that array variable names begin with @
• To access a single element in an array we use a scalar name and a number called the array index inside square brackets [ ] . Array indices begin with zero!
  • print the name of an author
  • print "Darwin's Radio was written by ", $items[0], "\n"

  • print the answer to the universe
  • print $items[1], "\n"

  • print the last item in the list using -1 as the index
Perl Arrays

• We can store a list of values in an array variable:
  • assign values to the @items array variable
  • @items = ("Greg Bear", 42, "X", 3.5e-107);

• Notice that array variable names begin with @
• To access a single element in an array we use a scalar name and a number called the array index inside square brackets [ ]. Array indices begin with zero!
  • print the name of an author
  • print "Darwin's Radio was written by ", $items[0], "\n";

  • print the answer to the universe
  • print $items[1], "\n";

  • print the last item in the list using -1 as the index
  • print $items[-1], "is a very small number! \n";
Adding/removing elements to/from Arrays
• We can add an item to the end of an array with the push function:
• We can add an item to the end of an array with the `push` function:
  • assign values to the `@items` array variable
We can add an item to the end of an array with the `push` function:

- assign values to the `@items` array variable
- `@items = ("Greg Bear", 42, "X", 3.5e-107);`
• We can add an item to the end of an array with the `push` function:
  • assign values to the `@items` array variable
  • `@items = ("Greg Bear", 42, "X", 3.5e-107);`
  • `push (@items, "Guster");`
Adding/removing elements to/from Arrays

- We can add an item to the end of an array with the `push` function:
  - assign values to the `@items` array variable
  - `@items = ("Greg Bear", 42, "X", 3.5e-107);`
  - `push (@items, "Guster");`

- We can remove (and retrieve) an item from the end of an array with the `pop` function:
Adding/removing elements to/from Arrays

• We can add an item to the end of an array with the push function:
  • assign values to the @items array variable
  • @items = ("Greg Bear", 42, "X", 3.5e-107);
  • push (@items, "Guster");

• We can remove (and retrieve) an item from the end of an array with the pop function:
  • $pop_band = pop (@items);
• We can add an item to the end of an array with the push function:
  • assign values to the @items array variable
  • @items = ("Greg Bear", 42, "X", 3.5e-107);
  • push (@items, "Guster");

• We can remove (and retrieve) an item from the end of an array with the pop function:
  • $pop_band = pop (@items);

• To add and remove items at the beginning of an array we can use the shift and unshift functions:
Adding/removing elements to/from Arrays

- We can add an item to the end of an array with the push function:
  - assign values to the @items array variable
  - @items = ("Greg Bear", 42, "X", 3.5e-107);
  - push (@items, "Guster");

- We can remove (and retrieve) an item from the end of an array with the pop function:
  - $pop_band = pop (@items);

- To add and remove items at the beginning of an array we can use the shift and unshift functions:
  - use shift to remove and retrieve the first array element
Adding/removing elements to/from Arrays

• We can add an item to the end of an array with the push function:
  • assign values to the @items array variable
  • @items = ("Greg Bear", 42, "X", 3.5e-107);
  • push (@items, "Guster");

• We can remove (and retrieve) an item from the end of an array with the pop function:
  • $pop_band = pop (@items);

• To add and remove items at the beginning of an array we can use the shift and unshift functions:
  • use shift to remove and retrieve the first array element
  • $author = shift (@items);
Adding/removing elements to/from Arrays

• We can add an item to the end of an array with the push function:
  • assign values to the @items array variable
  • @items = ("Greg Bear", 42, "X", 3.5e-107);
  • push (@items, "Guster");

• We can remove (and retrieve) an item from the end of an array with the pop function:
  • $pop_band = pop (@items);

• To add and remove items at the beginning of an array we can use the shift and unshift functions:
  • use shift to remove and retrieve the first array element
  • $author = shift (@items);

• now insert a different author's name as the first element
Adding/removing elements to/from Arrays

• We can add an item to the end of an array with the push function:
  • assign values to the @items array variable
  • @items = ("Greg Bear", 42, "X", 3.5e-107);
  • push (@items, "Guster");

• We can remove (and retrieve) an item from the end of an array with the pop function:
  • $pop_band = pop (@items);

• To add and remove items at the beginning of an array we can use the shift and unshift functions:
  • use shift to remove and retrieve the first array element
  • $author = shift (@items);

• now insert a different author's name as the first element
  • unshift (@items, "Kurt Vonnegut");
Stepping (Iterating) through Perl Arrays

@items = ("Greg Bear", 42, "X", 3.5e-107);
foreach my $element (@items) {
    print "Element is: $element \n";
    if ( $element == 42 ) {
        print "So long and thanks for all the fish!! \n";
    }
}

Stepping (Iterating) through Perl Arrays

• Often it’s useful to step through an array element by element and do things with each value. The foreach loop makes this easy to do:

```perl
@items = ("Greg Bear", 42, "X", 3.5e-107);
foreach my $element (@items) {
    print "Element is: $element \n";
    if ($element == 42) {
        print "So long and thanks for all the fish!! \n";
    }
}
```
Stepping (Iterating) through Perl Arrays

- Often it’s useful to step through an array element by element and do things with each value. The foreach loop makes this easy to do:

```perl
@items = ("Greg Bear", 42, "X", 3.5e-107);
foreach my $element (@items) {
    print "Element is: $element \n";
    if ($element == 42) {
        print "So long and thanks for all the fish!! \n";
    }
}
```
Stepping (Iterating) through Perl Arrays

• Often it’s useful to step through an array element by element and do things with each value. The foreach loop makes this easy to do:

```perl
@items = ("Greg Bear", 42, "X", 3.5e-107);
foreach my $element (@items) {
    print "Element is: $element \n";
    if ( $element == 42 ) {
        print "So long and thanks for all the fish!! \n";
    }
}
```

• The foreach (*) loop is nice because you don't need to worry about how many elements the array contains. The scalar value named after the foreach keyword refers to each element in turn and changes with each iteration of the loop.
Stepping (Iterating) through Perl Arrays

• Often it's useful to step through an array element by element and do things with each value. The foreach loop makes this easy to do:

```perl
@items = ("Greg Bear", 42, "X", 3.5e-107);
foreach my $element (@items) {
    print "Element is: $element \n";
    if ( $element == 42 ) {
        print "So long and thanks for all the fish!! \n";
    }
}
```

• The foreach (*) loop is nice because you don't need to worry about how many elements the array contains. The scalar value named after the foreach keyword refers to each element in turn and changes with each iteration of the loop.

• (*) In the Cozen's book for is used instead of foreach
print "Enter your name: ";
<STDIN>;
print "Your name is ";

@names = ("Bob", "Carol", "Ted", "Alice");
foreach (@names) {
    if ($_ eq "Alice") {
        print "You can have anything you want…\n";
        print "May I help you, $_?\n";
    } else {
        print "May I help you, $_?\n";
    }
}
Perl has a special variable named \$_ that gets assigned values by default if no other variable is explicitly named.

```
print "Enter your name: ";
<STDIN>;
print "Your name is \$_";

@names = ("Bob", "Carol", "Ted", "Alice");
foreach (@names) {
    if (\$_ eq "Alice") {
        print "You can have anything you want...\n";
        print "You can have anything you want...\n";
    } else {
        print "May I help you, \$_?\n";
    }
}
```
The special "default" variable \$_

- Perl has a special variable named \$_ that gets assigned values by default if no other variable is explicitly named.
- Examples:

```perl
print "Enter your name: ";
<STDIN>;
print "Your name is \$_";

@names = ("Bob", "Carol", "Ted", "Alice");
foreach (@names) {
    if (\$_ eq "Alice") {
        print "You can have anything you want...\n";
    } else {
        print "May I help you, \$_?\n";
    }
}
```
Other Array Functions: scalar and sort

- To find the number of elements in an array, use the scalar function:
Other Array Functions: scalar and sort

- To find the number of elements in an array, use the scalar function:
  - `@items = ("Greg Bear", 42, "X", 3.5e-107);`
• To find the number of elements in an array, use the scalar function:
  • @items = ("Greg Bear", 42, "X", 3.5e-107);
  • $num_items = scalar (@items);
Other Array Functions: scalar and sort

• To find the number of elements in an array, use the scalar function:
  • @items = ("Greg Bear", 42, "X", 3.5e-107);
  • $num_items = scalar (@items);

• You may also see $#array being used to get the index of the last element in an array:
Other Array Functions: scalar and sort

- To find the number of elements in an array, use the scalar function:
  - @items = ("Greg Bear", 42, "X", 3.5e-107);
  - $num_items = scalar (@items);

- You may also see $#array being used to get the index of the last element in an array:
  - @items = ("Greg Bear", 42, "X", 3.5e-107);
Other Array Functions: scalar and sort

• To find the number of elements in an array, use the scalar function:
  • @items = ("Greg Bear", 42, "X", 3.5e-107);
  • $num_items = scalar (@items);

• You may also see $#array being used to get the index of the last element in an array:
  • @items = ("Greg Bear", 42, "X", 3.5e-107);
  • $last_index = $#items;
Other Array Functions: scalar and sort

• To find the number of elements in an array, use the scalar function:
  • @items = ("Greg Bear", 42, "X", 3.5e-107);
  • $num_items = scalar (@items);

• You may also see $#array being used to get the index of the last element in an array:
  • @items = ("Greg Bear", 42, "X", 3.5e-107);
  • $last_index = $#items;

• It is very easy to sort an array using the sort function:
Other Array Functions: scalar and sort

- To find the number of elements in an array, use the scalar function:
  ```perl
  @items = ("Greg Bear", 42, "X", 3.5e-107);
  $num_items = scalar (@items);
  ```

- You may also see `@items` being used to get the index of the last element in an array:
  ```perl
  @items = ("Greg Bear", 42, "X", 3.5e-107);
  $last_index = $#items;
  ```

- It is very easy to sort an array using the sort function:
  ```perl
  @items = ("Greg Bear", 42, "X", 3.5e-107);
  ```
Other Array Functions: scalar and sort

- To find the number of elements in an array, use the scalar function:
  - @items = ("Greg Bear", 42, "X", 3.5e-107);
  - $num_items = scalar (@items);

- You may also see $#array being used to get the index of the last element in an array:
  - @items = ("Greg Bear", 42, "X", 3.5e-107);
  - $last_index = $#items;

- It is very easy to sort an array using the sort function:
  - @items = ("Greg Bear", 42, "X", 3.5e-107);
  - @sorted = sort (@items);
Other Array Functions: scalar and sort

- To find the number of elements in an array, use the scalar function:
  ```perl
  @items = ("Greg Bear", 42, "X", 3.5e-107);
  $num_items = scalar (@items);
  ```

- You may also see $#array being used to get the index of the last element in an array:
  ```perl
  @items = ("Greg Bear", 42, "X", 3.5e-107);
  $last_index = $#items;
  ```

- It is very easy to sort an array using the sort function:
  ```perl
  @items = ("Greg Bear", 42, "X", 3.5e-107);
  @sorted = sort (@items);
  print "The sorted items are @sorted \n";
  ```
The defined, exists functions for testing values
The defined, exists functions for testing values

• It is possible to declare a variable without defining it (assigning a value).
  Example:
The defined, exists functions for testing values

- It is possible to declare a variable without defining it (assigning a value).
  Example:
  ```perl
  my $name;  # Declared, not defined
  ```
The defined, exists functions for testing values

- It is possible to declare a variable without defining it (assigning a value).
  Example:
  ```perl
  my $name;    # Declared, not defined
  my $name = "Joe";    # Declared and defined
  ```
• It is possible to declare a variable without defining it (assigning a value). Example:

```perl
my $name;   # Declared, not defined
my $name = "Joe"; # Declared and defined
```

• The Perl defined function lets us check to see whether a variable has been given a value:
The defined, exists functions for testing values

- It is possible to declare a variable without defining it (assigning a value).
  Example:
  ```perl
  my $name;  # Declared, not defined
  my $name = "Joe";  # Declared and defined
  ```

- The Perl defined function lets us check to see whether a variable has been given a value:
  ```perl
  if (defined $name) {
  ```
The defined, exists functions for testing values

- It is possible to declare a variable without defining it (assigning a value). Example:
  ```perl
  my $name;  # Declared, not defined
  my $name = "Joe";  # Declared and defined
  ```

- The Perl defined function lets us check to see whether a variable has been given a value:
  ```perl
  if (defined $name) {
    print "Name is $name \n";
  }
  ```
The defined, exists functions for testing values

• It is possible to declare a variable without defining it (assigning a value).
  Example:
  
  my $name;  # Declared, not defined
  my $name = "Joe";  # Declared and defined

• The Perl defined function lets us check to see whether a variable has been given a value:
  
  if (defined $name) {
    print "Name is $name \n";
  }


The defined, exists functions for testing values

• It is possible to declare a variable without defining it (assigning a value).
  Example:
  
  ```perl
  my $name;  # Declared, not defined
  my $name = "Joe";  # Declared and defined
  ```

• The Perl defined function lets us check to see whether a variable has been given a value:
  
  ```perl
  if (defined $name) {
    print "Name is $name \n";
  }
  ```

• The exists function can be used to check whether an array element has been initialized:
It is possible to declare a variable without defining it (assigning a value).
Example:

```perl
my $name;  # Declared, not defined
my $name = "Joe";  # Declared and defined
```

The Perl defined function lets us check to see whether a variable has been given a value:

```perl
if (defined $name) {
    print "Name is $name \n";
}
```

The exists function can be used to check whether an array element has been initialized:

```perl
my @names = ("Bud", "Cal", "Doc", "Edd");
```
The defined, exists functions for testing values

• It is possible to declare a variable without defining it (assigning a value). Example:
  
  ```perl
  my $name;  # Declared, not defined
  my $name = "Joe";  # Declared and defined
  ```

• The Perl defined function lets us check to see whether a variable has been given a value:
  
  ```perl
  if (defined $name) {
    print "Name is $name \n";
  }
  ```

• The exists function can be used to check whether an array element has been initialized:
  
  ```perl
  my @names = ("Bud", "Cal", "Doc", "Edd");
  if (exists $names[3] && exists $names[4]) {
  ```
The defined, exists functions for testing values

• It is possible to declare a variable without defining it (assigning a value).
  Example:
  ```perl
  my $name;  # Declared, not defined
  my $name = "Joe";  # Declared and defined
  ```

• The Perl defined function lets us check to see whether a variable has been
given a value:
  ```perl
  if (defined $name) {
      print "Name is $name \n";
  }
  ```

• The exists function can be used to check whether an array element has been
  initialized:
  ```perl
  my @names = ("Bud", "Cal", "Doc", "Edd");
  if (exists $names[3] && exists $names[4]) {
      print "The name following $names[3] is $names[4] \n";
  }
  ```
The defined, exists functions for testing values

• It is possible to declare a variable without defining it (assigning a value).
  Example:
  ```perl
  my $name;  # Declared, not defined
  my $name = "Joe";  # Declared and defined
  ```

• The Perl defined function lets us check to see whether a variable has been given a value:
  ```perl
  if (defined $name) {
      print "Name is $name \n";
  }
  ```

• The exists function can be used to check whether an array element has been initialized:
  ```perl
  my @names = ("Bud", "Cal", "Doc", "Edd");
  if (exists $names[3] && exists $names[4]) {
      print "The name following $names[3] is $names[4] \n";
  }
  ```
The chomp function for removing newlines
The chomp function for removing newlines

When a line of text is read, it contains a newline character at the end. Often it is necessary to strip this character from the line, using the `chomp()` function.
• When a line of text is read, it contains a newline character at the end. Often it is necessary to strip this character from the line, using the `chomp()` function

```perl
$name = <STDIN>;
```
The chomp function for removing newlines

- When a line of text is read, it contains a newline character at the end. Often it is necessary to strip this character from the line, using the `chomp()` function.

```perl
$name = <STDIN>;
# Chomp off the newline
```
When a line of text is read, it contains a newline character at the end. Often it is necessary to strip this character from the line, using the `chomp()` function.

```perl
$name = <STDIN>;
# Chomp off the newline
chomp($name);
```
The chomp function for removing newlines

- When a line of text is read, it contains a newline character at the end. Often it is necessary to strip this character from the line, using the chomp() function

```perl
$name = <STDIN>;
# Chomp off the newline
chomp($name);
if ($name eq "Doc") {
```
The chomp function for removing newlines

- When a line of text is read, it contains a newline character at the end. Often it is necessary to strip this character from the line, using the chomp() function

```perl
$name = <STDIN>;
# Chomp off the newline
chomp($name);
if ($name eq "Doc") {
    print "It only hurts when I laugh... \n";
```
When a line of text is read, it contains a newline character at the end. Often it is necessary to strip this character from the line, using the chomp() function

```perl
$name = <STDIN>;
# Chomp off the newline
chomp($name);
if ($name eq "Doc") {
    print "It only hurts when I laugh... \n";
}
```
The chomp function for removing newlines

• When a line of text is read, it contains a newline character at the end. Often it is necessary to strip this character from the line, using the chomp() function

```perl
$name = <STDIN>;
# Chomp off the newline
chomp($name);
if ($name eq "Doc") {
    print "It only hurts when I laugh... \n";
}
```

• Don’t forget about chomp() – doing so often bites beginning Perl programmers!!!
The chomp function for removing newlines

- When a line of text is read, it contains a newline character at the end. Often it is necessary to strip this character from the line, using the chomp() function
  ```perl
  $name = <STDIN>;
  # Chomp off the newline
  chomp($name);
  if ($name eq "Doc") {
      print "It only hurts when I laugh... \n";
  }
  ```
- Don’t forget about chomp() – doing so often bites beginning Perl programmers!!!
- If your script is behaving strangely and you are reading an input file, there may be extra unprintable characters in the file. You can use the cat command with options –vet to reveal these, i.e.
When a line of text is read, it contains a newline character at the end. Often it is necessary to strip this character from the line, using the `chomp()` function

```perl
$name = <STDIN>;
# Chomp off the newline
chomp($name);
if ($name eq "Doc") {
    print "It only hurts when I laugh... \n";
}
```

Don’t forget about `chomp()` – doing so often bites beginning Perl programmers!!!

If your script is behaving strangely and you are reading an input file, there may be extra unprintable characters in the file. You can use the `cat` command with options –vet to reveal these, i.e.

```bash
cat -vet file.txt
```
Perl Flow Control: Looping with foreach

@names = ("Bud", "Cal", "Doc", "Edd");
foreach $n (@names) {
    print "Wassup $n ?\n";
    if ($n eq "Doc") {
        print "Ha Ha Ha Ha Ha Ha !!!! \n";
    }
}
The foreach loop is used to step through array elements. Below is an example that uses foreach and if. Notice the matched pairs of curly braces {} and the indentation in the code:

```perl
@names = ("Bud", "Cal", "Doc", "Edd");
foreach $n (@names) {
    print "Wassup $n ?\n";
    if ($n eq "Doc") {
        print "Ha Ha Ha Ha Ha !!!! \n";
    }
}
```
$n = 10;
while ( $n > 0 ) {
    print "Subtracting 2 from $n \n";
    $n = $n - 2;
    print "The result is $n \n";
}
A looping construct that is not tied to an array is the while loop. The code inside the while { ... } statement block is executed repeatedly, as long as the condition remains true.

```perl
$n = 10;
while ( $n > 0 ) {
    print "Subtracting 2 from $n \n";
    $n = $n - 2;
    print "The result is $n \n";
}
```
Perl Flow Control: Looping with while

• A looping construct that is not tied to an array is the while loop. The code inside the while { … } statement block is executed repeatedly, as long as the condition remains true.

• Example:

```perl
$n = 10;
while ( $n > 0 ) {
    print "Subtracting 2 from $n \n";
    $n = $n - 2;
    print "The result is $n \n";
}
```
foreach $n (@numbers) {
    if ($n == 0) { next; }  # Avoid division by zero
    $ratio = $value / $n;
    print "ratio is: $ratio \n";
}

foreach $n (@names) {
    print "Wassup $n \n";
    if ($n eq "Doc") {
        print "We got our man !!!! \n";
        last;  # exit the foreach loop
    }
}  # end of foreach name

• To jump to the next iteration of a loop, use `next`

```perl
foreach $n (@numbers) {
    if ($n == 0) { next; } # Avoid division by zero
    $ratio = $value / $n;
    print "ratio is: $ratio \n";
}

foreach $n (@names) {
    print "Wassup $n ?\n";
    if ($n eq "Doc") {
        print "We got our man !!!!! \n";
        last; # exit the foreach loop
    }
} # end of foreach name
```
To jump to the next iteration of a loop, use `next`

```perl
foreach $n (@numbers) {
    if ($n == 0) { next; }  # Avoid division by zero
    $ratio = $value / $n;
    print "ratio is: $ratio \n";
}
```

To jump out of a loop completely, use `last`

```perl
foreach $n (@names) {
    print "Wassup $n ?\n";
    if ($n eq "Doc") {
        print "We got our man !!!! \n";
        last;  # exit the foreach loop
    }
}
```

# end of foreach name
if ($total > 10e9 || $total < 10e3) {
    print "Unexpected total: $total \n";
}

if (defined $sum && $avg <= 33.3) {
    $result = 0;
}

# Be careful about operator precedence here!
if (defined $sum && $avg < 33.3 || $avg > 102.2) {
    $result = $sum - $avg;
}

if (!defined $total) {
    print "Total is undefined, cannot compute average\n";
}
The conditions in an if or while test can be simple, or complex (using && || !)

```perl
if ($total > 10e9 || $total < 10e3) {
    print "Unexpected total: $total \n";
}

if (defined $sum && $avg <= 33.3) {
    $result = 0;
}

# Be careful about operator precedence here!
if (defined $sum && $avg < 33.3 || $avg > 102.2) {
    $result = $sum - $avg;
}

if (!defined $total) {
    print "Total is undefined, cannot compute average\n";
}
```
The conditions in an if or while test can be simple, or complex (using && || !)

Examples:

```php
if ($total > 10e9 || $total < 10e3) {
    print "Unexpected total: $total \n";
}

if (defined $sum && $avg <= 33.3) {
    $result = 0;
}

# Be careful about operator precedence here!
if (defined $sum && $avg < 33.3 || $avg > 102.2) {
    $result = $sum - $avg;
}

if (!defined $total) {
    print "Total is undefined, cannot compute average\n";
}
```
Perl Hashes (Associative Arrays)

- We’ve seen arrays and the use of integers as index values: $items[0], $items[1], etc.
Perl Hashes (Associative Arrays)

• We’ve seen arrays and the use of integers as index values: $items[0], $items[1], etc.

• Sometimes it is useful to store <Key, Value> pairs rather than using integers to index an array
Perl Hashes (Associative Arrays)

- We’ve seen arrays and the use of integers as index values: $items[0], $items[1], etc.

- Sometimes it is useful to store <Key, Value> pairs rather than using integers to index an array.

- Perl Hashes do just that. Another name for a Hash is an Associative Array.
Perl Hashes (Associative Arrays)

- We’ve seen arrays and the use of integers as index values: `$items[0]`, `$items[1]`, etc.

- Sometimes it is useful to store `<Key, Value>` pairs rather than using integers to index an array.

- Perl Hashes do just that. Another name for a Hash is an Associative Array.

- You can build a ‘dictionary’, containing keywords and definitions associated with these keywords.
Perl Hashes (Associative Arrays)

- We’ve seen arrays and the use of integers as index values: $items[0], $items[1], etc.

- Sometimes it is useful to store <Key, Value> pairs rather than using integers to index an array.

- Perl Hashes do just that. Another name for a Hash is an Associative Array.

- You can build a ‘dictionary’, containing keywords and definitions associated with these keywords.

- Hash syntax is similar to array syntax, but employs different symbols.
# define species key, value pairs
%species = ('human' => 'H.sapiens',
            'mouse' => 'M.musculus',
            'fruitfly' => 'D.melanogaster');

print $species{'mouse'}, "\n";
Perl Hash Example

- Array variables begin with the @ sign, and to index an individual item, use [[]]: @arr = (1,3,5); $arr[3] = 7;

```perl
# define species key, value pairs
%species = ('human' => 'H.sapiens',
            'mouse' => 'M.musculus',
            'fruitfly' => 'D.melanogaster');

print $species{'mouse'}, "\n";
```
Perl Hash Example

• Array variables begin with the @ sign, and to index an individual item, use [@]: @arr = (1,3,5); $arr[3] = 7;

• Hash variables begin with the % sign. Key,value pairs are connected with the double arrow =>

```perl
# define species key, value pairs
%species = ('human' => 'H.sapiens',
            'mouse' => 'M.musculus',
            'fruitfly' => 'D.melanogaster');

print $species{'mouse'}, "\n";
```
Perl Hash Example

- Array variables begin with the `@` sign, and to index an individual item, use `[]`: `@arr = (1,3,5); $arr[3] = 7;`

- Hash variables begin with the `%` sign. Key,value pairs are connected with the double arrow `=>`

- To index an individual item, use `$hash{'key'}`

```perl
# define species key, value pairs
%species = ('human' => 'H.sapiens',
            'mouse' => 'M.musculus',
            'fruitfly' => 'D.melanogaster');

print $species{'mouse'}, "\n";
```
Perl Hash Example

• Array variables begin with the @ sign, and to index an individual item, use [ ]: @arr = (1,3,5); $arr[3] = 7;

• Hash variables begin with the % sign. Key,value pairs are connected with the double arrow =>

• To index an individual item, use $hash{'key'}

• Example:

```perl
# define species key, value pairs
%species = ('human' => 'H.sapiens',
            'mouse' => 'M.musculus',
            'fruitfly' => 'D.melanogaster');

print $species{'mouse'}, "\n";
```
Adding to and Removing from a Hash
Adding a key, value pair to a hash is easy. Of course, each key must be distinct.
Adding a key, value pair to a hash is easy. Of course, each key must be distinct.

Example:
Adding a key, value pair to a hash is easy. Of course, each key must be distinct.

Example:

```perl
$species{‘blowfish’} = 'T.rubripes';
```
Adding a key, value pair to a hash is easy. Of course, each key must be distinct.

Example:

$species{‘blowfish’} = 'T.rubripes';

Removing a key, value pair from a hash is done by the delete function.
Adding to and Removing from a Hash

- Adding a key, value pair to a hash is easy. Of course, each key must be distinct.
  - Example:
    - $species{'blowfish'} = 'T.rubripes';

- Removing a key, value pair from a hash is done by the delete function.
  - Example:
Adding a key, value pair to a hash is easy. Of course, each key must be distinct.

Example:

```perl
$species{'blowfish'} = 'T.rubripes';
```

Removing a key, value pair from a hash is done by the delete function.

Example:

```perl
delete $species{'human'};
```
Adding a key, value pair to a hash is easy. Of course, each key must be distinct.

Example:

```
$species{‘blowfish’} = 'T.rubripes';
```

Removing a key, value pair from a hash is done by the delete function.

Example:

```
delete $species{‘human’};
```

The exists function can be used to check for existing hash entries.
Adding a key, value pair to a hash is easy. Of course, each key must be distinct.

- Example:
  - `$species{`blowfish`} = `T.rubripes`;`

Removing a key, value pair from a hash is done by the delete function.

- Example:
  - `delete $species{`human`};`

The exists function can be used to check for existing hash entries.

- Example:
Adding a key, value pair to a hash is easy. Of course, each key must be distinct.

- Example:
  - `$species{‘blowfish’} = 'T.rubripes';`

Removing a key, value pair from a hash is done by the delete function.

- Example:
  - `delete $species{‘human’};`

The exists function can be used to check for existing hash entries.

- Example:
  - `if (exists $species{‘human’}) { ... }`
Looking up Keys or Values in a Hash
Looking up Keys or Values in a Hash

• You can get a list of all values in a hash using the values function:
• You can get a list of all values in a hash using the `values` function:

  ```perl
  @values = values (%hash);
  ```
You can get a list of all values in a hash using the values function:

```perl
@values = values (%hash);
```

The `values()` function takes a hash as an argument and returns an array of values.
You can get a list of all values in a hash using the values function:

```
@values = values (%hash);
```

The `values()` function takes a hash as an argument and returns an array of values.

Similarly, a list of all keys in a hash can be obtained by using the keys function:
Looking up Keys or Values in a Hash

- You can get a list of all values in a hash using the `values` function:
  - `@values = values (%hash);`
- The `values()` function takes a hash as an argument and returns an array of values.

- Similarly, a list of all keys in a hash can be obtained by using the `keys` function:
  - `@keys = keys (%hash);`
• You can get a list of all values in a hash using the `values` function:
  • `@values = values (%hash);`

• The `values()` function takes a hash as an argument and returns an array of values.

• Similarly, a list of all keys in a hash can be obtained by using the `keys` function:
  • `@keys = keys (%hash);`

• The `keys()` function takes a hash as an argument and returns an array of values.
while ( my ($key, $value) = each(%hash) ) {
    print "$key => $value\n";
}

foreach my $key ( keys %hash ) {
    my $value = $hash{$key};
    print "$key => $value\n";
}
To step through each key, value pair in a hash, use a foreach loop and the keys function:

```perl
while ( my ($key, $value) = each(%hash) ) {
    print "$key => $value\n";
}

foreach my $key ( keys %hash ) {
    my $value = $hash{$key};
    print "$key => $value\n";
}
```
Stepping through Key, Value pairs in a Hash

• To step through each key, value pair in a hash, use a foreach loop and the keys function:

```perl
while ( my ($key, $value) = each(%hash) ) {
    print "$key => $value\n";
}
```

• TIMTOWTDI:

```perl
foreach my $key ( keys %hash ) {
    my $value = $hash{$key};
    print "$key => $value\n";
}
```
To step through each key, value pair in a hash, use a foreach loop and the keys function:

```perl
while ( my ($key, $value) = each(%hash) ) {
    print "$key => $value\n";
}
```

TIMTOWTDI:

```perl
foreach my $key ( keys %hash ) {
    my $value = $hash{$key};
    print "$key => $value\n";
}
```

Note that hash elements are not ordered!