Perl generalizations

Whitespace is generally insignificant

- spaces, tabs, newlines

Comments will start with the pound sign (#) and continue to the end of the line

- Shebang notation is a special type of comment that may be used by the operating system

The "main" program consists of everything that isn't within a function or subroutine

Variables to not have to be declared before their use

Most statements end with a semicolon

main package

All Perl programs are compiled into a package

If the package is not named the default name is main.

- main is never actually mentioned in the file
- All code not in a subroutine is considered to be in the main package

All packages provide their own namespace for variables

- Almost all variables within a package are global within that package

Literals

Literals are just constants of any type

Integer literals can be expressed as positive or negative values in decimal, hexadecimal, octal or binary format

Floating-point literals can be represented in floating-point or scientific notation

String literals are usually delimited by either single or double quotes

Numeric literals

Floating point literals can be expressed several different ways

- 100.0, 1e4, 1.23e-4, .314159E+1

Non-decimal integer literals can be used

- A leading 0 mean octal
  - 0377 is 255 decimal
- A leading 0x means hexidecimal
  - 0x20 is 32 decimal
- A leading 0b means binary
  - 0b00001111 is 15 decimal

Underscores (_) can be used in any numeric literal and are ignored by Perl

- 123456.78 is the same as 123_456.78

Data types and variables

Perl tends to classify variables by how much data type hold, not the type of data

3 basic data types

- Scalar – holds a single value
  - Will start with a $ sign
- Arrays – holds a list of scalar values
  - Will start with an @ sign
- Hashes – sometimes called an associative array
  - Will start with a % sign
  - Similar to an array; consists of one or more pairs of scalars

Variable names

Variables do not have to be declared in Perl

All variables must start with one of the 3 characters ($, @, %)

Variable names are case sensitive

Variable names (after the leading character) may contain any number of letters, numbers, and underscores (_) as long as they start with a letter or underscore

The leading character counts as part of the variable name so $team is a different variable than @team or %team and all 3 are legal

Scalars

A single variable is called a scalar - variable starts with a $ character

A scalar is the fundamental type from which more complicated structures are built

Stores a single value, usually a string or a number

- Values of different types are used interchangeably for many operations

Numeric assignment statements

- Assignment statements work like every other language

```
$variable = expression;
```

  - An expression can be a literal, another variable or some equation using both literals and variables

- Everything is computed using double-precision floating point values – no built in integer type
  - Range and precision are based upon the underlying architecture

String assignment statements

Single quoted strings are assigned to the variable exactly as written

Double quoted string literals are assigned to variable after variable interpolation and escape sequence substitution

Curly brackets { } can be placed around the text portion of the variable name to separate the variable name from other characters

```
$vehicle = “motorcycle”;
print “He owns several $vehicles\n”; #doesn’t work
print “He owns several ${vehicle}s\n”;  #does work
```

Arithmetic operators and functions

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`%`	modulus
`**`	exponent

Shortcut assignment operators

`+=, -=, *=, /=, %=, **=`

To truncate a floating-point value to an integer value use the `int()` function.

```
$intValue = int($floatingPointValue);
```

String operators and functions

.

concatenate strings

x

repeat the first string z times

```
$twentyBlanks = “ “ x 20;
print ">$twentyBlanks<\n";
```

Shortcut operators

.=

x=

`substr ($str1, $offset, $len)`

returns substring of $len bytes from $str1 starting at position $offset

`index ($str1, $str2)`

Returns the byte offset of string $str2 in string $str1

Other string functions

`length (expression)`

Returns the length, in bytes, of the expression

`rindex ($str, $substr, position)`

Returns the position of the last occurrence of $substr in $str

Position is optional; if specified start at that byte, otherwise start at the end of the string

`chr(number)`

Returns the character specified by the ASCII value of the argument

`ord(character)`

Returns the ASCII value of the character specified

`lc($str)`

Returns a lowercase string

`uc($str)`

Returns an uppercase string

Mixing data types

Perl automatically converts between numbers and string as needed

Never concern yourself with whether a value of a variable holds a number of strings – it's both

If a numeric operator is used on a string value, the numeric value of the string is computed

- Trailing non-numeric character(s) or leading whitespace is ignored
- Leading zero(s) in a string does not make the value be interpreted as octal

If a string operator is used on a numeric value the number is converted to a string

True and False

There is no boolean type in Perl

Scalar values are interpreted as true or false (if used in a context that requires a boolean value)

The following expressions evaluate to false

```
“”
‘’
()
“0” (or ‘0’)
0
undef
```

All other expressions evaluate to true

Variable interpolation

Variable names placed inside a double quoted string are replaced by their value when printed or used to assign a value to a string variable

Escape sequences are interpreted in a double-quoted string (just a short list of available sequences)

\t	tab
\n	newline
\r	carriage return
\\	backslash
\'	single quote
\"	double quote
\$	dollar sign

Variable interpolation does not occur in single-quoted strings

Printing information

The print function prints a comma-separated list of expressions

By default this information is sent to the standard output device (STDOUT), which is the screen

Strings are normally delimited by either a matching pair of double (") or single (') quotes.

When using single quotes all the characters are treated as literals.

- No variable substitution
- No escape character substitution

When using double quotes most of the characters are treated as literals

- Variable substitution and escape characters occur in double quoted strings

<STDIN>

Line input operator

```
$input = <STDIN>;
```

Accepts input from the terminal

- Causes program execution to pause while the computer waits for the user to enter data

Always returns characters up to and including the next newline character

Reads data into the special variable `$_` if no variable is assigned to `<STDIN>` (if in a while statement)

Chop, chomp

The chop function removes the last character from a string and returns that character

```
chop ($string); #ignores the chopped off character
```

Uses $_ as the argument if none is specified

```
$lastChar = chop($string2);   #chops off the last
  character and stores it in $lastChar
```

The chomp function removes the last character off the end of a string if the character matches the contents of the special variable $/ (input record separator)

Uses $_ as the argument if none is specified

$/ defaults to the newline character

$/

CHANGE WITH CARE

Stores the default character for chomp function

Also used with `<STDIN>`. `<STDIN>` reads until the `$/` character is read. BY DEFAULT this will read until the newline is entered.

Backquotes

The backquote or backtick is the quoted execution operator

Backquotes (`) are used to execute UNIX/Linux/Windows commands

Output from these commands is returned to the Perl program

- Variables are interpolated before control is transferred to the shell
- Output can be printed or assigned to a variable

```
$dir = `pwd`
print "Current directory is $dir";
```

- pwd is a UNIX/Linux command to return the current directory
- cd is the windows command to return the current directory

Quote operators

q(…) or q/../ can be used in place of single quotes

'string' is the same as q/string/ or q(string)

qq(…) or qq/…/ can be used in place of double quotes

"string" is the same as qq/string/ or qq(string)

qx(…) or qx/…/ can be used in place of backquotes

`cd` is the same as qx/cd/ or qx(cd)

qw(…) or qw/…/ can be used to create a single quoted, comma delimited list of strings

‘a’, ‘b’, ‘c’, ‘d’ is the same as qw/a b c d/ or qw(a b c d)

More on Quote Operators

When using q, qq, qx, or qw any non-alphanumeric, non-whitespace delimiter can be used in place of the /

- Basically “syntactic sugar” to simplify placing backslashes into quotes string
- If the opening delimiter is a parenthesis, bracket, brace or angle bracket the closing delimiter will be the matching construct.
- Embedded occurrences of the delimiters must match in pairs

A word that has no other interpretation in the grammar will be treated as if it were a quoted string. These are known as barewords.

Warnings

Pragmas are statements that the compiler uses to set compiler options

- The `use` command is one option for setting various pragmas

Add to source code

```
use warnings;  #warning pragma
```

Or add –w to perl statement

```
#!/usr/bin/perl –w
```

Or command line

```
perl –w perlScript
```

Warns (among other things) that variables are being used that don't have a value

- Other than assigning those variables a value
- Also includes possible typos

Strict

Add to source code

```
use strict;  #strict pragma
```

Forces the programmer to declare all variables as package or lexically scoped variables (more later), to quote all strings and to call each subroutine explicitly

Strict can also be passed tag arguments to tell the compiler what it should check

- use strict ‘tag’
- Possible tags
  - ‘vars’, ‘subs’, ‘refs’

To temporarily turn off strict use

```
no strict;
```

```
#to turn off checking of just references
no strict 'refs'
```

Constants

The use constant statement will create a scalar whose value cannot be changed. The use of all capital letters is not required but recommended.

```
use constant CONSTANT_NAME => value;
use constant PI => 3.14159;
```

Defining a constant happens at compile time.

It's not a good idea to put a constant declaration inside of a conditional statement.

```
if ($whatever) { use constant ... }  #bad idea
```

Constants do not interpolate into double-quoted strings.

```
print "The value of PI is ", PI, ".\n";
```

Declaring variables

To create "local" variables in the sense of most common languages use the `my` keyword followed by a list of variable names.

```
my (variable1, variable2, variable3);
```

The `use vars` statement is followed by a list containing the names of variables to use.

```
use vars qw($var1 $var2 $var3 );
use vars ('$var1', '$var2', '$var3');
```

printf function

Allows the programmer to generate formatted output

Every printf function call contains a format-control string that describes the output

- May include format specifiers

%d	decimal
%f	floating-point values
%s	string
%e	scientific notation
%c	single character
%%	prints a percent sign

- Each format specifier may include a minimum width field width and number of places after the decimal point for floating-point values.
- Values are automatically right justified; a leading – (minus sign) immediately after the % will left justify any type of value

%10d	Right justifies an integer value within 10 spaces
%10.2f	Right justifies a floating-point value within 10 spaces with 2 digits after the decimal point
%-10s	Left justifies a string within 10 spaces

printf and sprintf capabilities

Rounding floating-point values to a specified number of decimal places

- Floating point values are automatically rounded, not truncated for info not displayed

Aligning a column of numbers with decimal points

Right and left justification of output

Displaying floating-point numbers in exponential form (e)

Displaying integers in octal and hexadecimal form (o and x)

Displaying all numeric values with fixed-size widths and precision

sprintf takes the same arguments as printf (except for the optional filehandle) but returns the requested string instead of printing it

printf examples

The format-control string is followed by a comma and an expression for every format specifier

```
printf ("format string with embedded specifiers",
        comma-separated variable list – one for each specifier;

printf "Integer value with leading zeros %05d", $integerVariable;

printf "Floating value %8.2 string value %8s",
        $floatVariable, $stringVariable;

printf "Left justified integer %-8d\nLeft justified float on next line %-5.1f",
        $integerVariable, $floatVariable;

my $money = sprintf "%.2f", $someUnformattedAmount;
```

here doc string

here-doc is a special way of printing out large amounts of text

Can be done in any Perl program

here doc strings begin with << followed by an identifier and end when that identifier appears on a line all by itself somewhere later in the program.

- Here-doc markers are traditionally in all caps (not required)
- Here-docs are useful for quoting large passages of text or source code
- If the marker is quoted with single quotes no variable interpolation is performed
- If the identifier is quoted with double quotes variable interpolation is performed – double quote mode is the default
- The ending marker must be on a line containg the marker followed by a newline – nothing else can be on the line

```
print <<EndOfHereDoc  # or print <<"EndOfHereDoc"
text to be output with $variables
more text to be output
EndOfHereDoc
```

undef

undef is the value of variables not assigned a value

- Neither a number not a string
- Distinct type
- Used whenever a value is undefined
- All scalar variables start with the value of undef before their first assignment
- Looks like zero if used as a number or the empty string if used as a string
  - Easy to write numeric accumulators that start as if they were zero
  - Easy to write string accumulators that start as if they were the empty string
- This includes scalars variables and unused elements of an array

undef also can undefine an already defined variable and release whatever memory was allocated for the variable

- undef $name;

Many operators return undef to indicate failure or boundary condition

defined

- defined function allows you to check for the validity of a variables value
  - defined ($variableName) returns 1 if the variable has a value and null if it doesn't have a value
- defined function can be used to determine the difference between a variable that stores undef and an empty string

Lists

A list is a collection of values. To create a list separate a series of values with commas and enclose the list in parenthesis.

List context

A section of code in which a list is required for the program to complete it's task

When a list appears on the left side of an assignment operator whatever is on the right hand side of the operator is evaluated in a list context

Scalar context

A section of code in which a scalar is required for the program to complete it's task

A scalar variable on the left side of an assignment operator gives scalar context to what is on the right hand side of the operator

Arrays

An array in Perl stores a list of scalars - variable starts with an @ character

The list may hold any combination or scalar values: strings, numbers or a mixture

The individual elements of the array can be accessed using numeric subscripts placed inside brackets [ ]

Subscripts are integers

Subscripts start at 0

Use a $ to access the elements of the array – the elements are scalar values

A negative index starts counting from the end of the array

Arrays and lists

Arrays and lists are not the same thing.

An array is a data structure
- Permanently allocated memory
- Generally named

A list is a collection of values
- Temporary list of values stored on the runtime stack

Arrays and lists are usually interchangeable
- Perl converts lists and arrays back and forth as needed
  - A list assigned to an array variable becomes an array
  - An array used in a list context becomes a list of values

- They are not always interchangeable
  - Lists cannot be popped

Assigning values to arrays

Easiest way to initialize an array is to assign a list to the array

Elements in the list are simple scalars

```
@names = (“Tom”, “Bob”, “Scott”, “Roy”);
@names = qw/Tom Bob Scott Roy/; #another way to do it
@names = ( );                  #empties the names array
```

This creates the scalar variables

```
$names[0]    # value of Tom – also referenced by $names[-4]
$names[1]    # value of Bob – also referenced by $names[-3]
$names[2]    # value of Scott – also referenced by $names[-2]
$names[3]    # value of Roy – also referenced by $names[-1]
```

More on assigning values to arrays

Arrays can grow as needed and are limited by the computers memory

```
@names = (“Tom”, “Roy”, “Scott”, “Bob”);   #creates elements 0 thru 3
$names[9] = “Brian”;
#creates elements 4 thru 8 and assigns those elements the value undef
```

Array values can be copied from one array to another

```
@names2 = @names;            #names2 is a full copy of names
```

Range operator (..)

Used for the creation of a consecutive range of string or numeric values

```
@values = (1 ..10);
@letters = (‘a’ .. ‘z’);
@values2 = (1..10, 20, 30..40);
```

Special array scalars

$#arrayname returns the number of the last subscript in the array

- Can also be used to shorten the array

$[is the current array base subscript

- Defaults to 0
- Can be changed but it’s highly recommended to not do so

scalar function

- The scalar function forces the argument to the function to be evaluated in a scalar context
- Using scalar (@arrayname) returns the total number of elements in @arrayname

Array slices

When the elements of one array are assigned to elements of another array it's called an array slice

If there are less scalar variables in the list on the left side of the assignment operator than there are on the right side only the number of needed values are taken

If an array appears on the left side of a list assignment enough elements in the array are created to store all the remaining elements

If there are more variables in the list on the left side of the assignment operator than there are values to assign the extra variables receive the value undef

Array slices are generally consecutive elements of an array but do not have to be

The bracket operator, [ ], can be used to create a list containing a specified set of elements from an array

Array slices

A slice has the characteristics of a list of variable names

```
($first, $second, $third) = @sortedArrayOfScalars;
($sortedItems[0], $sortedItems[1])=@sortedArrayOfScalars;
#same as previous line
@sortedItems[0, 1] = @sortedArrayOfScalars;
```

A slice used in a scalar context returns the last value in the slice

STDIN with arrays

When reading into an array using STDIN each line is read and treated as a single list item

- Data is read until Ctrl-D (UNIX) or Ctrl-Z (Windows) is pressed
- The newline is included with each element in the array

Chop / chomp for arrays

Chop takes one argument, an array in this context, and removes the last letter of each string in the array

```
chop (@arrayname);
```

Chomp takes one argument, an array in this context and removes the last character from each string in the array if it is a newline. Chomp returns the number of newlines removed from the array.

```
chomp (@arrayname);
$noOfNewlinesRemoved = chomp (@arrayname);
```

Pop / push

Pop takes one argument, an array, and removes the last element from an array and returns that element and reduces the size of the array by one

```
pop @arrayname;
pop (@arrayname);
$poppedItem = pop (@arrayname);
```

Push takes two arguments; an array and a list of elements to be inserted at the end of the array

push (@array, $element) is identical to $array[@array] = $element

Shift / unshift

Shift takes one argument, an array, and removes the first element from an array and returns that element and reduces the size of the array by one

```
shift @arrayname;
shift (@arrayname);
$shiftedItem = shift (@arrayname);
```

Unshift takes two arguments; an array and a list of elements to be inserted at the beginning of the array

```
unshift (@array, $element);
unshift (@array, $firstElement, $secondElement);
unshift (@array, @otherArrayToGoInFrontOfTheOriginalArray);
```

join function

$scalarVariable = join (separator, list)

The Join function will join each of the individual elements in a list (array) into one string. Each element of the list will be separated by the separator string

Separator can be any string but generally contains a comma, colon or other field divider

Returns a single scalar value

split function

split (delimiter, string, limit)

- The split function scans a string and splits it into separate fields. A field is composed of every character between two delimiters (also called field separators).
- Returns a list of scalars
- The function returns the number of fields if used in a scalar context
- All arguments are optional going from right to left
  - If no limit it works until the end of the string
  - If the string is omitted (along with the limit) $_ is used as the string
- The delimiter can be in quotes or apostrophes
  - Searches for the string specified; quoted strings use variable interpolation
- The delimiter can be a regular expression if placed in slashes (more on regex's later :-)
  - / / looks for a single blank space

sort

The sort function sorts and returns a list (in ASCII order)

The original list is left unchanged

Arguments are computed early so the result can be assigned back into the same variable

```
@names = qw/Tom Bob Scott Roy/;
@sorted = sort (@names);
```

To change the sort order code must be provided that compares two elements in the list to determine their sorting order. This is placed in curly brackets after the sort statement but before the list to sort

```
sort { code to do the sort } @list;
```

The code must compare two elements called $a and $b and return 1 is $a is greater than $b, return 0 if $a and $b are equal and return -1 is $a is less than $b

```
@sorted = sort {$a cmp $b} @names;  # same as earlier example
@numSorted = sort {$a <=> $b} @names; #sorts numerically
```

reverse

The reverse function takes a list as an argument and returns a new list with the same contents in reverse order

- The original list is left unchanged

- Arguments are computed early so the result can be assigned back into the same variable

- @reverse = reverse @sorted;

splice

splice (array, offset, length, list)

- array – array to modify
- offset – the index of the first element to modify in the array
- length – length of the slice to modify
- list – list to replace the specified slice of the array

The splice function removes or replaces slices of an array

All but the first argument are optional

- If the list is omitted the slice is simply removed from the array
- If the list and the length are omitted the slice from the offset to the end of the array is removed
- If the offset (and length and list) are omitted all the array elements are removed

If the return value is another array it will consist of those elements removed

If the return value is a scalar value the returned value is the last element removed

Multi-dimensional arrays

Multidimensional arrays are lists of lists

They consist of rows and columns and are represented by multiple subscripts

Each row in a two-dimensional array is enclosed in square brackets

The arrow operator or infix operator (->) can be used to access individual elements in an array

Hashes

- Hashes are similar to arrays
  - Start with a % character
  - Indices, also knows as keys, are arbitrary unique strings
    - Keys serve the same purpose as indexes with arrays
    - Using a “word” as an array index can add more meaning or understanding to the code
  - The info stored for each element in the hash are called values
  - Implemented using a hash-table so they are fast and stay fast regardless of their size
  - Unlike arrays the elements have no particular order – just a collection of key-value pairs

More on Hashes

Hashes can be of any size

- Limited only by memory
- New elements in the hash are created automatically

A hash must be defined before the elements can be referenced

When using a hash curly braces are used instead of square brackets

Keys are always converted to strings

- An expression can be used as the key
- Keys may be in quotes but the quotes may be omited if the key is a valid Perl identifier

Pairs in a hash can appear in any order

Hash assignment

Hashes can be assigned a value with a list

```
%honorPoints = (“A”, 4, “B”, 3, “C”, 2,
                “D”, 1, “F”, 0);
```

- The odd numbered items in the list are the keys
- The even numbered items in the list are the values

The digraph operator can be used within a list to indicate keys/values

```
%honorPoints = ( A => ‘4’, B => ‘3’, C => ‘2’,
                 D => ‘1’, F => ‘0’);
```

- The digraph operator (=>) is a synonym for the comma operator. The only difference in functionality is when the => operator is used the argument to the left of => is always treated as a string (will not be interpreted as a function call).

Keys / values

- The keys function returns an array whose elements are the keys of a hash
  - The return value of the keys function in a scalar context is the number of key/value pairs in the hash
- The values function returns an array consisting of the values of a hash
  - The return value of the values function in a scalar context is the number of key/value pairs in the hash

each

The each function returns one key-value pair where the first element is the key and the second element is the value of a hash

The each function keeps track of where it is in the hash so every call to this function returns a new key-value pair

- Returns undef when there are no more pairs to return

```
%honorPoints ( A => ‘4’, B => ‘3’, C => ‘2’,
               D => ‘1’, F => ‘0’);
($letter, $points) = each (%honorPoints);
```

exists / delete

The exists function returns true if an array index (or hash key) has been defined and false if it has not

The delete function removes a value from an element of an array, but not the element itself

- The value of the deleted item is undefined

Arithmetic and string operators

Arithmetic operators

+	addition
-	subtraction
*	Multiplication
/	division
%	modulus
**	exponent

String operators

.	concatenate strings
x	repeat the first string z times

All operators have a shortcut option

Relational operators

Perl has two class of relational operators – one for numbers and one for strings

Numeric relational operators

- `>` greater than
- `>=` greater than or equal
- `<` less than
- `<=` less than or equal
- `==` equal
- `!=` not equal
- `<=>` numeric comparison with a signed return

  $val1 <=> $val2

  returns 1 if $val1 is greater than $val2

  returns 0 if $val1 and $val2 are equal

  returns –1 if $val2 is greater than $val1

Relational operators continued

String relational operators

gt	greater than
ge	greater than or equal
lt	less than
le	less than or equal
eq	equal
ne	not equal
cmp	ASCII comparison with a signed return

$val1 <=> $val2

returns 1 if $val1 is greater than $val2

returns 0 if $val1 and $val2 are equal

returns –1 if $val2 is greater than $val1

Logical operators

&&	
and	logical and
||	
or	logical or
xor	exclusive or
!	
not	logical not

Perl logical operators are also known as short circuit operators

Perl does not return 0 or 1 for false and true. It returns the value of the last operand evaluated

Perl evaluates the operands from left to right and stops evaluations once a true or false condition is guaranteed

- If the expression to the left of the && operator evaluates to 0 (false) no further evaluation is done
- If the expression to the left of the || operator evaluates to 1 (true) no further evaluation is done

Autoincrement / autodecrement

Perl has taken the ++ and -- operators from C

Java also took the ++ and -- operators from C

If used within an assignment or combined with other operators the leading or trailing operator will have a different effect on the outcome

If used in an expression

The preincrement operator (++$scalar) will add 1 to the variable and then use that value in the expression

The postincrement operator ($scalar++) will use the value of scalar in the expression and then add 1 to the variable

The predecrement and postdecrement operators work the same way

Bitwise operators

&	bitwise and
|	bitwise or
^	bitwise exclusive or
<<	bitwise left shift, integer multiply by 2
>>	bitwise right shift, integer divide by 2

```
$x = 4;

$y = $x & 2;
$z = $x | 2;

$yy = $x && 2;
$zz = $x || 2;
```

Condition operators

The conditional operator (?:) is taken from the C language

The condition operator is a ternary operator, meaning it takes three operands

- Works as a shortcut for the if/else statement

(logical expression) ? expression1 : expression2;

- If the logical expression is true the value of expression1 is returned
- If the logical expression is false the value of expression2 is returned