Subroutines / Functions

- Perl doesn’t distinguish between the term function and subroutine
- Subroutines are global and may be placed anywhere in the script
  - Can be in the same file or in a different file
- All variables used within a program, and it’s subroutines/functions are automatically global variables and have global scope unless declared otherwise
  - Variables can be specifically declared in a lexical or dynamic scope

Creating subroutines

The general format of a subroutine is

```
sub subroutineName
{
   # subroutine block
}
```

Must start with the keyword sub

subroutineName can be any valid identifier

Subroutines can receive optional arguments. Arguments passed to the subroutine are received into the special array variable @_

@_

Arguments are not specifically “received” into variables

- No parameter list of variables can be specified

Any number of arguments can be passed into a subroutine

Changing the values in the @_ array changes the value of the actual parameters

- The elements of the @_ array are just aliases for the values passed in
- In general you should start off a subroutine by copying the arguments from the array into "local" variables
- Arguments, either scalars or arrays, are passed using the call-by-reference mechanism
- Shifting or popping elements from the @_ array just looses the reference to those parameters, the values are not lost in the calling program

Inside a subroutine, shift uses @_ as a default argument.

- Outside a subroutine @ARGV is used as a default argument for `shift`

Data “flattening”

Data is “flattened” as it is received into @_

If you send an array and a scalar as arguments @_ will store a single list of data starting with the values from the array followed by the value of the scalar

Hashes passed as an argument are received as a list of key/value pairs (key1, value1, key2, value2, etc.)

There are things that can be done to avoid flattening of the data (check using References)

Returning a value

When a subroutine completes it's task, data is returned with the `return` keyword

The return statement causes the subroutine to stop executing immediately

If no return statement is specified the value of the last expression evaluated (either a scalar or an array) is returned

The return statement can return a scalar or an array

- The wantarray() function can be used within the subroutine to determine the type of data that is expected to be returned
  - Returns true if an array is expected, returns false if a scalar is expected

Invoking a subroutine

Normally subroutines are called by following the subroutine name with a set of parenthesis. Arguments are provided inside the parenthesis

The physical order in which the subroutine and calling the subroutine occur affect how the subroutine is called

Subroutines can be called by preceding the subroutine name with an ampersand (&)

- If no arguments are specified the parenthesis can be omitted
- Parenthesis are always required if arguments are passed to the subroutine

Bareword syntax – not using special symbols around the subroutine name help Perl figure out it's a subroutine call

- If called after the subroutine is written then the subroutine is invoked
- If called before the subroutine is written Perl interprets the bareword like a string and doesn't invoke the subroutine

Local variables

Perl variables are global unless otherwise specified

- Keyword **our** can be used to declare variables
- **our** was added as of version 5.6

A lexically scoped variable is a variable that exists only during the block in which it was defined

- Uses the keyword **my** to declare variables
- Creates “local” variables in the sense of other languages
- my is the suggested way to create “local” or temporary variables

A dynamically scoped variable is a variable that exists from the time it is created until the end of the block in which it was created

- Uses the keyword **local** to declare variables
- These variables are global within the program
- "run-time" scoping – saves value of existing variable; then restores the value when leaving the containing scope
- special variables – any $-punctuation variable – can only be localized with `local`; it is an error to attempt to localize a special variable with my

my example

```
sub max
{
   # variables can be created and
   # assigned in the same step
   my ($a, $b) = @_;
   if ($a > $b)
      { $a }
   else
      { $b }
}
```

References

References are scalar variables which hold a value that tells them where to find another value in memory

- A reference indirectly points to a value
- Pointing to a value using a reference is called indirection
- Hard references to variables are created by preceding the name of the variable with the unary backslash operator (\)
  - There are also symbolic, or soft references that we won't cover

```
$x = 10;
$xRef = \$x;    #xRef is a reference to $x
```

- To get the value back you must dereference the reference by using its entire name (including the $ that precedes it) like a normal variable

```
print "$$xRef\n";     #prints value of $x
```

More references

References can be created for any kind of data structure – just precede the nonscalar variable name with the unary backslash operator

\@array returns a reference to an array @array
\%hash returns a reference to a hash %hash

To dereference an individual element of an array

```
$$array[$indexNo]
$array->[$indexNo]
```

To reference the entire array

```
@$array
```

To reference a hash element

```
$$hash{‘key’}
$hash->{‘key’}
```

To reference the entire hash

```
%$hash
```

Dereferencing arrays and hashes

To dereference arrays and hashes that have been passed as arguments to a subroutine place the appropriate leading character in front of the scalar passed into subroutine

```
#calling the sub
&subName ( \@array, \@hash);

#in the sub
sub subName
{
   my @localArray = @{$_[0]};
   my %localHash = %{$_[1]};
```

The curly braces are required to force the order of evaluation because the precedence of the @ sysmbol is high than the square brackets

- Without the curly braces the @ symbol would deference $_ and the square brackets would make an array slice of element 0

Anonymous References

A value need not be contained in a defined variable to create a reference.

To create an anonymous array reference use square brackets instead of parenthesis

```
$arrayRef = [20, 30, 50, 80];
```

@$arrayRef can be used anywhere an array (or list) is expected

For hash references, use curly brackets, instead of parenthesis

```
$hashRef={“sky”=>‘blue’,“grass”=>‘green’}
```

%$hashRef can be used anywhere a hash is expected

Dereferencing specific elements of references

How many ways can we deference specific elements from

```
$arrayRef = [20, 30, 50, 80];
```

or

```
$hashRef = {sky => ‘blue’, grass => ’green’ };
```

TMTOWTDI

In fact, there are three…

```
$arrayRef = [20, 30, 50, 80];
$$arrayRef[2] = 60;
${$arrayRef}[2] = 60;
$arrayRef->[2] = 60;

$hashRef = {sky=>'blue',grass =>
  'green'};
$$hashRef{sky} = $value;
${$hashRef}{sky} = $value;
$hashRef->{sky} = $value;
```

These are all valid and acceptable. The form you choose is whatever looks the best to you.

Determining the type of a reference

The ref function takes a reference as an argument. Using in a boolean context ref returns a true value is the argument is a reference, otherwise it returns a false value.

Used in other than a boolean context the value returned depends upon the type of thing the reference is a reference to. The built in return values include

REF
SCALAR
ARRAY
HASH
CODE
GLOB

Soft references

Soft references are scalar objects containing strings representing identifiers of variables and functions. Soft references are also called symbolic references.

By placing a variable inside the curly braces (also known as lookups) the value of the scalar is taken to be the name of the variable

```
$var1 = 1;
$var2 = 'var1';
print ${$var2}, "\n";  #displays 1
print ${'var' . $var1} , "\n";  #also displays 1
```

Only global and local variables are available for soft references

Soft references are very powerful and can be dangerous

- To display soft references use `use strict 'refs'`
- To enable soft references use `no strict 'refs'`

What to know about scope

my is statically (lexically) scoped

- Look at the actual code. Whatever block encloses **my** is the scope of the variable

local is dynamically scoped

- The scope is the enclosing block, plus any subroutines called from within that block

Almost always want **my** instead of **local**

- notable exception: cannot create lexical variables such as **$_**. Only ‘normal’, alpha-numeric variables
- for built-in variables, localize them.

hashes for passing named parameters

Typeglobs

References are a recent addition to Perl. Before references typeglobs were used

To pass a variable as it's type rather than having it's data flattened place a * in front of the variable rather than the standard symbol ($, @, %) for it’s data type when passing the variable as an argument

Whatever type the argument was when passed is retained when accessed from from the subroutine

A single data item is passed for the array or hash rather than n items, where n is the length of the list

To reclaim the original format of the argument use a * in front of the variable when getting the argument from the argument list (@_).

- There is no way to tell what type each element of the @_ was when passed. The program must know what to expect.
- After assigning a value to a local variable you now use the standard symbol for it’s original data type
- local variables must be used to accept typeglobs in the subroutine

Predeclaring a subroutine

A declaration tells the compiler that a subroutine will be written

It may include the type of arguments that must be included (prototyping)

```
sub subRoutineName;
```

Prototypes

Perl's way of letting you limit how you'll allow your subroutine to be called.

When defining the function, give it the 'type' of variable you want it to take in parenthesis before the opening curly brace

```
sub f1 ($$) {…}
#f1 must take two scalars

sub f2($@) {…}
#f2 takes a scalar, followed by a list

sub f3(\@$) {…}
#f3 takes an actual array, followed by a scalar
```

Prototype generalities

if prototype char is:	Perl expects:
\$	actual scalar variable
\@	actual array variable
\%	actual hash variable
$	any scalar value
@	Array or list – ‘eats’ rest of params and force list context
%	Hash or list – ‘eats’ rest of params and forces hash context
*	file handle
&	subroutine (name or definition)

Getting around parameters

If you want to ignore parameters, call subroutine with & character in front

```
sub myfunc (\$\$){
    …
}

myfunc (@array); #ERROR!

&myfunc (@array); #No error here
```

references to subroutines

References can be made to subroutines as well as data types

To make a reference to a subroutine use the & operator in front of the subroutine name

```
$subRef = \&subroutine;
```

To deference the subroutine (and call the subroutine) use the & operator in front of the scalar reference variable

```
$subReturnedValue = &$subRef
```

or

```
$subReturnedValue = &$subRef (arguments)
```

To invoke a subroutine-type sub (rather than a function-type sub) just call the deferenced sub

&$subRef;

Or

&$subRef (arguments);

Packages

Packages are basically named libraries, or collections of code

Perl uses packages, or namespaces, to determine the scope of variables and subroutines.

- If a file contains a package the package is usually fully contained in that file, but this is not a requirement
  - While not common a file can contain more than one package or a package can be spread out over multiple files
- Packages are similar to classes in Java or C++
- Each package has its own symbol table
  - The symbol table is implemented with a hash the with the variable name being the key

By default all variables are global in Perl

- Actually the variables are package global variables

The “main” program is called the main package. If a name is not given to a package it defaults to main.

Creating packages

By default a program has one package, the main package.

Other packages can be created by using the package statement. Packages provide a separate namespace so that you can avoid naming collisions.

To create a new package the statement

```
package packageName;
```

tells the compiler that everything that follows for the remainder of the file (or until the next package statement) are in the packageName package

Writing the package

Start with the package statement

Any variables or subroutines declared will part of that package

Every variable or subroutine referred to within this package will be expected to be within this package.

The package must end with a true (non-zero) value for it to be used by the compiler

```
1;  # traditional last line of a file
```

using Perl packages

One option to bring a package into a program is to use the **require** statement

```
require "fileNameOfFileWithPackage"
```

or

```
require “fileNameOfFileWithPackage.pl”;
```

If no file extension is specified with the filename it is assumed to be .pm, otherwise the full filename plus extension must be used. If no extension is specified the filename can be a bareword

Perl will first look in the current directory for the file. If the file cannot be found there Perl searches for directories that are named in the array @INC

- @INC is a built-in array with preset values determined by the installation directory

Accessing elements of a package

To refer to anything in the package use a fully qualified name, in which the package and the variable name or subroutine name are joined by the double colon operator (::)

- This tells the Perl compiler which symbol table it should into to locate an identifier

```
packageName::subroutineName();

$packageName::scalarVariable;
```

- Note that the $ required for the scalar reference goes before the packageName

Modules

A module is a package that provides the programmer more control over how a user of the module can reference the identifiers in that module's package. The purpose of modules is to implement reusable, modular functionality.

To be a module a package must

- be contained in a separate file whose name ends in .pm
  - Only one package can be defined per file
  - The extension of the saved file must be pm, not pl.
- define an import method – usually done by subclassing the Exporter module
  - Allows variables within the module to be made available to the user as if they were in the main program
- define a (possibly empty) list of symbols that are automatically exported, as well as a (possibly empty) list of symbols that can be exported on request

Modules are generally stored in a library directory that is accessible via Perl's default include path

Using Perl modules

The use statement is used to load modules

- The use statement will only load files with the pm extension

`use packageName;`

- Note that the .pm and the quotes are excluded from the packageName when using the use statement

`require` vs. `use`

- The main difference between use and require is that use imports modules and packages at compile time and require does so at execution time.

- Importing at compile time allows Perl to ensure that the package is available before the program reaches execution time.
  - Allows Perl to determine that a package is missing before it actually referenced

Placing subroutines in different packages

Subroutines can be “placed” into other packages by preceding the subroutine name with the target package and a apostrophe

```
package packageName;

sub main’subName { #whatever the sub does }
```

subName is placed into the main package and can be referred to by only subName is the main program

- The subroutine cannot be referred to by the packageName::subName even though it physically exists in packageName

Variables can be placed in other packages as well

```
$main’scalarVar = 0;
```

Exporting package members

Identifiers and subroutines can be exported from one package to another namespace. This allows the other file to use the identifiers and subroutines without the fully qualified names. To do this the module must be set up as an Exporter module.

```
use Exporter;
```

This indicates that the current modules uses the Exporter module

```
our @ISA = qw ( Exporter );
```

The special built-in array @ISA needs to contain the Exporter module

```
our @EXPORT = qw ( @array $scalar %hash
  &subroutine);
```

The built-in array @EXPORT needs the names of the identifiers and subroutines to be exported to a different namespace

@INC array

The INC array contains a colon-separated set of directories where Perl looks to find required files. You can place a library file in any directory as long as you add this directory to the @INC array.

To add a directory to the @INC array use either the push or unshift function

```
push (@INC, “c:/perl/myLibs”);
```

Or

```
unshift (@INC, “c:/perl/myLibs”);
```

To make a permanent change to the directories in the @INC array use the PERL5LIB environment variable. If the variable exists Perl will check it and check the specified directories. Changing the PERL5LIB settings is operating system specific.

- colon separated list in *NIX systems
- semicolon separated list in Windows systems

Adding to the include path

The lib pragma was added in Perl 5.

To prepend one or more directories to the include path use the path(s) as arguments to use lib. Adds to the paths search for modules for the execution of this program – not a permanent change.

```
use lib "/new/Path/To/Module";
use lib qw(
   c:/a/WindowsPath
   /and/one/more/Linux/Path
);

use ModuleFoundInAPathSpecifiedAbove;
```

BEGIN blocks

A BEGIN block is used for code that is to be executed immediately after it is compiled; and before any following code is compiled

```
BEGIN
{
   # code to be executed at compile time
}
```

Since the BEGIN blocks are compiled and executed before the "normal" code it usually doesn't matter where a BEGIN block is placed within the program

- It will be executed only once at compile time
- A program may contain multiple BEGIN blocks

BEGIN blocks may contain my variables and subs

END blocks

END blocks are used for code that will be executed just as a Perl program terminates. END blocks are useful for cleaning up – closing files, closing database connections, deleting temporary files, things of that nature

```
END
{
   # termination code
}
```