COMS 3101
Programming Languages: Perl

Lecture 5

Fall 2013
Instructor: Ilia Vovsha

http://www.cs.columbia.edu/~vovsha/coms3101/perl

Lecture Outline

- Packages & Modules
- Concepts:
  - Subroutine references
  - Symbolic references
  - Garbage collection
  - Saving structures
- Objects and Classes
- Next: More OOP, CPAN

Remarks

- Pattern matching “cage” can be any character:
  - m// or // is equivalent to m{}
  - s/// is equivalent to s{} {}
  - // are just customary quote characters for pattern matching behavior. In fact you could choose your own character instead of {} (e.g. m ! !)
  - Convenient if lots of slashes in the pattern
- ref function: returns type of reference (a string)
  - $rtype = ref($href); # returns “HASH”
  - $rtype = ref($aref); # returns “ARRAY”
  - if (ref($href) eq “HASH”) { … }

Packages & Modules

- Why do we need them?
- Package or module, what is the difference?
- ‘use’ vs. ‘require’
- Importing from another package
- Pragmatic modules (pragmas)

Packages (purpose)

```
sub  parse_text  {                              # code from one file
    …
    $count  =  $count++;
    …
}

sub  normalize  {                               # code from another file
    $count++;
    …
}

# Use both functions:
parse_text();
normalize();
print   “$count\n”;                             # Which $count? What is its value?
```

Packages (definition)

- Balanced code: abstraction + reuse
- Every chunk of code has its own namespace. In Perl, a *namespace* is called a *package*
- Independent of files:
  - Multiple packages in one file
  - Single package spanning multiple files
  - Most common: one package per file
- Best approach: one package per file where file name is package name + extension ‘.pm’
- ‘pm’ = perl module
- Each package has its own symbol table
- Default current package: ‘main’

Packages (example)

```
package  Parse;                                # namespace: Parse
sub  parse_text  {
     $count  =  $count+1;
     …
}
package  Process;                              # namespace: Process
sub  normalize   {                             # code from another file
     $count++;
     …
}

package  main;                                 # Implicit if no package declaration
Parse::parse_text();
Process::normalize();
print   “$Parse::count\n”;                          # count from Parse package
print   “$count  ,  $main::$count\n”;               # global variable count
```

Modules (definition)

- Fundamental unit of code reusability in Perl
- Module: one package in one file where file name is package name + extension '.pm'
- Two types:
  - Traditional: define vars/subs for caller to import / use
  - Object oriented: specify class definitions, and accessed through method calls
- To include module in code: 'use' or 'require'
- Naming convention: upper case letters

Modules (example)

```perl
# code.pl
use  Parse;
use  Process;

Parse::parse_text();
Process::normalize();
print  “$Parse::count\n”;
```

```perl
# Parse.pm
package  Parse;
$count  =  0;
sub  parse_text  {
    $count  =  $count+1;
    …
}
1;    # use must succeed!
```

```perl
# Process.pm
package  Process;
$count  =  0;
sub  normalize  {
    $count++;
    …
}
1;
```

Modules (*use vs. require*)

- To include module in code: ‘use’
- Preload at compile-time: compiler loads module before compiling rest of file (can change behavior, visibility)
- Imports requested symbols (Exporter module):
  - Specify explicitly
  - Import symbols from @EXPORT and @EXPORT_OK
  - Once imported, can be used without package name qualifier
- Access imported vars/subs with “::”
  - Process::normalize();   # With qualifier
  - normalize();   # No qualifier
- Searching for modules:
  - In each directory listed in @INC array
  - Modify @INC at compile time with lib pragma
- ‘require’ loads module at run-time
- Actually, no compelling reason to prefer ‘require’ over ‘use’

Importing from Package

- Modules export symbols by inheriting the ‘import’ method from the Exporter module
- Symbols exported by default: @EXPORT
- Symbols exported by request: @EXPORT_OK

```
# Parse.pm
package  Parse;
use  Exporter;
our  @ISA  = (“Exporter”);        # Inheritance

our  @EXPORT  =  qw($count  &parse_text);
our  @EXPORT_OK  =  qw(%h  @arr);

sub  parse_text   { …}
1;
```

```
# code.pl
use  Parse;
# use Parse qw(%h @arr)

parse_text();
%Parse::h  = ();
```

Pragmas

- Pragmatic modules (pragmas) are hints to compiler
- Only work with ‘use’/ ‘no’ (seen at compile time)
- Convention: names in lowercase letters
- Lexically scoped (just like *my* variables), effects limited to enclosing block
- Invoke: ‘use <pragma>’
- Disable: ‘no <pragma>’

Pragmas (usage)

- Recall, to show warnings: **#!/usr/local/bin/perl -w**
- Pragmas are preferable

```
use   warnings;
#  Enable warnings till end of file
...
{
        no    warnings;
        # Disable warnings till end of block
        ...
}
#  warnings are enabled back
```

Pragmas (examples)

- *warnings*: Perl complains about variables that are used only once, variable re-declarations, improper conversions etc.

```
use   warnings;
use   warnings   qw(io  syntax);
```

- *strict*: Perl is strict about what is legal code (subs,vars,refs)
  - vars: variable must be predeclared
  - refs: can’t use symbolic references
  - subs: “barewords” are syntax errors   (must quote strings)

```
use   strict;                      # all three
use   strict   “vars”;             # variables only
```

Pragmas (more examples)

- *constant*: declare named symbol an immutable constant, requires separate declaration for each symbol

```
use  constant  ARR_SIZE  =>  100;
use  constant  ARR_REF   =>  [1,2,3,4];

# Can’t interpolate ARR_SIZE into string (no $ sign in front)!
```

- *lib*: add directories to default search path, at compile-time

```
use  lib “/myperl/code/”;          # Add to @INC
no   lib “/yourperl/code/”;        # Delete from @INC

# You should delete only directories you added
```

Concepts

- Subroutine references
- Copying referent
- Symbolic references
- Garbage collection
- Saving structures

Subroutine References

- Reference to a sub follows similar rules
- Can use backslash or anonymous

```
sub  max  { …. }                      # Anonymous
$sub_ref   =  \&max;                  $sub_ref   =  sub  { …. };

# Calling subroutine
&$sub_ref();
$sub_ref -> ();
$sub_ref -> (1,4,6,8);

# Call sub using string
my   %cmds   =   ( “process” => \&process_data, “clear” => \&clear_data,
                   “any”  =>  sub   {….}    );
$cmds{$str} -> ();
```

Copy Referent

- To copy referent for another reference:

```
$aref1  = [ 1,2,3 ];
$aref2  = [ 4,5,6 ];

$aref2  = $aref1                    # Not a copy!  Rather refers to the same location!

$aref2  = [ @$aref1 ];              # Create a new anonymous reference
$href2  = {  %{$href1 } };          #  Same approach for hashes
```

Symbolic References

- Refers to the name of a variable
- If we try to dereference, we get access to variable!
- Can be dangerous! (*use strict ‘refs’* to prohibit)

```
$N =  “V”;
$$N  =  5;                    # Set scalar $V = 5
$N -> [0]  =  6;              # Set element 0 of array V to 6
$N -> {key}  =  7;            # Set key of hash V to 7
&$N;                          # Call sub  V
```

Garbage Collection

- High-level language: don’t worry about de-allocating memory
- Garbage collection = automatic reclamation process
- Block exited: locally scoped variables are freed up
- Don’t hide your garbage! (circular references)
- Using multiple large hashes in the same scope? Loop through the keys and ‘delete’ each one
- For objects: use *destroy* methods

Saving Data Structures

- Save any DS to use later: *Data::Dumper* module
- Turn DS into a string, save externally to a file, read in the file and recover DS with eval / do
- CPAN module Storable: more efficient, but cannot be shared across different architectures

```
use  Data::Dumper;
open (OUTFILE,  “> filename”);
print  OUTFILE  Data::Dumper->Dump([\%hash],  [‘*hash’]);
open (INFILE, “< filename”);
undef  $/;                        # undef record separator, read entire file
eval   <INFILE>;                  # recreate %hash
```

```
use  Storable;
store(\%hash,   “filename”);
$href   =  retrieve(“filename”);
%hash  =  % { retrieve(“filename”)  } ;
```

OOP

- Review of concepts: objects, classes, methods
- Object Oriented Programming in Perl
- Method invocation
- Object construction:
  - constructor
  - bless function
  - initialization
  - destructor
- Inheritance

OOP (concepts)

- Program: collection of objects
- Object: data structure with collection of behaviors
- Object: instance of a class
- Class: defines attributes (variables) and methods (functions), i.e. behavior applied to class/instances
- Each instance of class (object) keeps track of its own attribute values
- Instance methods refer to actions of specific object
- Class methods refer to actions of entire class of (many) objects
- To generate a new object we use a constructor method

OOP (more concepts)

- Can share methods between classes: inheritance
- Derived / sub class inherits methods from base / parent / super class
- The derived class can "update" the behavior of the parent class
- Given an (derived) object, how do we select the most appropriate method for it? Polymorphism
- General design principle: when using the object, the object is a black box (we shouldn't manipulate attributes / methods directly)
- Encapsulation: access objects through methods alone

OOP in Perl

- Perl supports OOP, but does not enforce the “rules”: you can break encapsulation
- Not the best choice for extensive OOP projects (not native). OOP is slower than a non-OOP solution
- Can often write good code without using all OOP techniques
- Supports OOP techniques: single/multiple inheritance, method overriding, destructors, operator overloading
- No special syntax for objects:
  - Objects are references / referents
  - Classes are packages (usually modules)
  - Methods are subroutines

OOP (method invocation)

- To access an object indirectly, invoke method
- Invocation:
  - locate sub (method) determined by class of invocant and method name
  - Call the sub, passing invocant as its first argument
  - Explicit invocation: using arrows …. INVOCANT->METHOD()
- Invocant can be package name or reference (to object)
  - package name: 1st argument to class method is class name (string)
  - reference: 1st argument to instance method is reference (to object)

```
$mazda  =  Car -> new();                    # Class: Car,  Method:  new, Object: $mazda
$mazda -> drive(“slow”);                    # Instance: $mazda, Method: drive

$method = “new”;                            # Don’t know method name ahead of time
$mazda  =  Car -> $method();
```

OOP (constructor)

- Note: method’s package is resolved at run-time (don’t know invocant)
- Note: regular sub’s package is resolved at compile-time
- Constructor is just a subroutine….err…method (typically invoked by the package name) that returns an object reference

```
# Car.pm
package  Car;

sub  new  {                               # Constructor
   my  $invocant  =  shift;               #  1st argument is always invocant
   my  $self  =  { };                     # Reference to empty hash
   bless ($self, $invocant);
   return  $self;                         # Return reference to object
}
…
```

OOP (bless function)

- To work on an object, reference must be “marked” with its class name
- The act of “marking” (turning ref into object ref) is called blessing
- bless function: 1st argument is ref, 2nd argument is class name to bless into (default: current package)

```
# Car.pm
package  Car;

sub  new  {                                        # Constructor (hides the bless)
  my  $invocant  =  shift;
  my  $self  =  { };
  bless ($self, $invocant);                        # bless (REF, “Car”);
  return  $self;
}
…
```

OOP (initialization)

- Typical approach: use reference to an anonymous hash (but could be any type of reference)
- Hash can be non-empty: maintain internal information (attributes) which are only manipulated by the object’s methods
- bless function: 1st argument is ref, 2nd argument is class name to bless into (default: current package)

```
# Car.pm
package  Car;

sub  new  {
  my  $class  =  shift;
  my  $self  =  { @_ };
  bless ($self, $class);
  return  $self;
}
…
```

```
# Call:  $lexus  =  Car -> new(maker=>“lexus”);
sub  new  {
  my  $class  =  shift;
  my  $self  =  {
        maker => “mazda”,
        color => “black”,
        @_,
   };
   ….
}
```

OOP (destructor)

- Destructors are rarely needed in Perl (automatic memory management)
- Define a subroutine DESTROY (required name, unlike new)
- Explicitly calling DESTROY is possible but seldom needed

```
# Car.pm
package  Car;

sub  new  { …. # Constructor}
sub  DESTROY  {                    # Destructor
   my  $self  =  shift;
   # Attend to filehandles, databse connections
   …
}
```

OOP (inheritance)

- We can define a hierarchy of classes to share methods between them
- Derived / sub class inherits methods from base / parent / super class
- The super-classes are specified in the @ISA array (declared with our) of the derived class. Each element of the array is a package (class) name
- Single Inheritance: one parent class (search parent for methods)
- Multiple Inheritance: multiple parent classes (search left-to-right through @ISA array, depth-first order)
- Once method is found, store in cache for efficiency
- If @ISA is changed, the cache is invalidated (avoid it if you can!)
- Constructor Inheritance: no automatic call to base class constructor

OOP (method search)

- If method not defined in class: search @ISA at runtime when a call is made
- Search order given multiple parent classes: left-to-right through @ISA array, depth-first

```
# Nissan.pm
package  Nissan;
our  @ISA  =  (Car, Truck);

# Call:  Nissan->new()->build(“sedan”);
# Search order:
1. Check if method (build) is defined in class (Nissan)
2. Check if method is defined in “$Nissan::ISA[0]” (Car class)
3. Check if method is defined in @Car::ISA   (Car’s @ISA array)
4. When done with ISA[0], repeat steps 2,3, for ISA[1] (Truck class)
```

Scoped Variables

- my
  - creates private variable visible only within block
  - hidden from outside of enclosing scope, and hides previously declared variables with identical name
  - confines *name & value* to scope
  - suitable for scalar/array/hash variables
- our
  - confines *name* only to scope (no effect on visibility)
  - suitable for scalar/array/hash variables
  - used to access global variables, their initial value inside block unchanged
  - effects or assignment persist after the scope of declaration (block)
- local
  - confines *value* only to scope
  - suitable for scalar/array/hash + more variables
  - initial value for variable is () or undef
  - value of variable is restored no matter how you exit the block (changes thrown away)
  - “dynamic” scope: value of variable depends on scope & changes during run-time
  - ‘my’ is preferable over ‘local’

Scoped Variables (example)

```
$office  =  "global";                     # Global $office
&say();                                   # prints "global"
&barney();                                # prints "barney global", lexical scope;
&fred();                                  # prints "fred fred", dynamic scope,
&say();                                   # prints "global", restored after &fred()

sub  say  { print  "$office\n";  }        # print the $office

sub  barney  {
       my   $office  =  "barney";
       print  "$office ";  &say();   }

sub  fred  {
       local $office = "fred";
       print "$office ";  &say();   }
```