Day 13: Error Handling and Debugging

Suggested Reading: Programming Perl (3rd Ed.), Chapter 20: The Perl Debugger
Reminders

• Turn in homework at **START** of class
• Writing code is **fun**!
  – Write at least a little every day
  – The more you do, the easier it gets
• When in doubt, ask questions!
The Condor Philosophy

• Errors *Will* Occur!
  – Murphy was an optimist
  – Anything that can go wrong, will go wrong.
  – Do your best to prevent errors
  – Must handle as best as possible when they do occur
  – You take, Zathras die. You leave, Zathras die. Either way, it is bad for Zathras.
Types of Errors

• Software bugs
  – Your software
  – Other software that yours interacts with
  – Never in my software, though!

• Hardware failures

• Bad data

• User error

• Network failures
Buggy Script

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

sub Print( $$ )
{
    my( $key, $ref ) = @_;
    print "$key: " . join( "", "$ref" ) . "\n";
}

my %Data = ( foo  => [ 1, 2, 3, 4 ],
             bar  => [ 0, 3, 5, 9 ],
             blah => [ 1 ], );
foreach my $key ( keys %Data ) {
    Print( $key, $Data{$key} );
}
```
Running it...

• So, we try to run it, and Perl tells us that it hates it

bash-3.2$ perl example-01.pl
Can't use an undefined value as an ARRAY reference at example-01.pl line 8.
bash-3.2$
#! /usr/bin/env perl
use strict;
use warnings;

sub Print( $$ )
{
    my( $key, $ref ) = @_; 
    print "<$key> <$ref>\n"; 
    print "$key: " . join( "", "", @{$ref} ) . "\n";
}

my %Data = ( foo => [ 1, 2, 3, 4 ],
            bar => [ 0, 3, 5, 9 ],
            blah => [ 1 ],
            keys => %Data );

foreach my $key ( keys %Data )
{
    Print( $key, $Data{key} );
}
bash-3.2$ perl -d example-01.pl

Loading DB routines from perl5db.pl version 1.28
Editor support available.

Enter h or `h h' for help, or `man perldebug' for more help.

main::(example-01.pl:12):       my $Data = (  
main::(example-01.pl:13):         foo => [ 1, 2, 3, 4 ],  
main::(example-01.pl:14):         bar => [ 0, 3, 5, 9 ],  
main::(example-01.pl:15):         blah => [ 1 ],  
main::(example-01.pl:16):       );

DB<1>
Debuggers...

- Debuggers capabilities:
  - Set break points
  - Step through code line by line
  - Examine variables
  - Watch points
  - Stack traces
Perl Debugger Help

List/search source lines:  Control script execution:
  l [ln|sub] List source code   T  Stack trace
  - or . List previous/current line s [expr] Single step [in expr]
  v [line] View around line    n [expr] Next, steps over subs
  f filename View source in file <CR/Enter> Repeat last n or s
  /pattern/ ?patt? Search forw/backw r Return from subroutine
  M Show module versions       c [ln|sub] Continue until position
  o . . . Set debugger options t [expr] Toggle trace [trace expr]
  <[=][{[}][}]>[> ] [cmd] Do pre/post-prompt b [ln|event|sub] [cmd] Set breakpoint
  ! [N|pat] Redo a previous command B ln|* Delete a/all breakpoints
  H [-num] Display last num commands a [ln] cmd Do cmd before line
  = [a val] Define/list an alias A ln|* Delete a/all actions
  h [db_cmd] Get help on command w expr Add a watch expression
  h h Complete help page        W expr|* Delete a/all watch exprs
  [[]]db_cmd Send output to pager ![] syscall Run cmd in a subprocess
  q or ^D Quit                   R  Attempt a restart
  Data Examination: expr  Execute perl code, also see: s,n,t expr
  x|m expr  Evals expr in list context, dumps the result or lists methods.
  p expr  Print expression (uses script's current package).
  S [!!]pat List subroutine names [not] matching pattern
  V [Pk [Vars]] List Variables in Package. Vars can be ~pattern or !pattern.
  X [Vars] Same as "V current_package [Vars]". i class inheritance tree.
  y [n [Vars]] List lexicals in higher scope <n>. Vars same as V.
  e  Display thread id E Display all thread ids.

For more help, type h cmd_letter, or run man perldebug for all docs.
#!/usr/bin/env perl
use strict;
use warnings;

sub Print( $$ )
{
    my( $key, $ref ) = @_; 
    print "$key: " . join( ",", ",", @{$ref} ) . "\n";
}

my %Data = ( foo => [ 1, 2, 3, 4 ],
             bar => [ 0, 3, 5, 9 ],
             blah => [ 1 ], );

foreach my $key ( keys %Data )
{
    Print( $key, $Data{$key} );
}
Exceptions

• Exception handling is a programming language construct ... designed to handle the occurrence of exceptions, special conditions that change the normal flow of program execution.

Exceptions in Python and Ruby

• Python uses `try / except` mechanism
  – Similar to many other languages which support exceptions

• Ruby uses a "rescue" block
#! /usr/bin/env python
import sys

if len(sys.argv) != 2:
    print >>sys.stderr, "usage: example <file>"
    exit(1)

fname = sys.argv[1]
try:
    fh = open( fname )
    for line in fh.readlines():
        print line,
    fh.close()
except Exception, e:
    print "Failed to open", fname, ":\", e
    exit(1)
Ruby Exception Example

```ruby
#!/usr/bin/env ruby

unless ARGV.size == 1
  warn "usage: example <file>"
  exit 1
end

file = ARGV[0]

begin
  IO.foreach(file) do |line|
    puts line
  end

rescue
  $stderr.print "IO failed: " + $!
  raise
end
```
Exceptions In Perl

• Perl has a crude exception mechanism
  – Somewhat a "bolt on"
  – Use an "eval" block to execute the code, and then examine $@ to determine if an exception occurred
Perl's eval()

- `eval EXPR or eval BLOCK`
  - In the first form, the return value of `EXPR` is parsed and executed as if it were a little Perl program. The value of the expression (which is itself determined within scalar context) is first parsed, and if there weren't any errors, executed in the lexical context of the current Perl program, so that any variable settings or subroutine and format definitions remain afterwards. Note that the value is parsed every time the "eval" executes. If `EXPR` is omitted, evaluates `$_`. This form is typically used to delay parsing and subsequent execution of the text of `EXPR` until run time.

- In the second form, the code within the `BLOCK` is parsed only once—at the same time the code surrounding the "eval" itself was parsed—and executed within the context of the current Perl program. This form is typically used to trap exceptions more efficiently than the first (see below), while also providing the benefit of checking the code within `BLOCK` at compile time.

Source: `perldoc -f eval`
Perl Exception Example

```perl
#!/usr/bin/env perl
use strict; use warnings;
die "usage: example <file>" unless scalar(@ARGV) == 1;
my $file = shift(@ARGV);
eval {
    open(FILE, $file ) or 
    die "Unable to open file '$file';
};
if ($@) {
    print "Ahh: $@";
}
else {
    while( <FILE> ) {
        print;
    }
    close( FILE );
}
print "bye\n";
```
Loading Optional Modules

```perl
#!/usr/bin/env perl
use strict;
use warnings;
sub Load($$) {
    my( $name, $feature ) = @_; 
    eval "require $name";
    if( $@ ) {
        print STDERR "Can't find $name: $feature disabled;\n";
    } else {
        import $name;
    }
}

Load( "BadModule", "Something stupid" );
Load( "Time::ParseDate", "Date parsing" );
```