Day 7: Correctness

Suggested Reading:

Programming Perl (3rd Ed.)
Chapter 20: The Perl Debugger

http://perldoc.perl.org → Modules
Test::Simple, Test::More, Test::Harness

Homework Review

Correctness

I wrote a script!

\o/

Is it right?

>_<

What Makes Software *Right*?

functionality	Does it do the correct thing?
reliability	Does it work every time?
usability	Is it easy and effective to use?
efficiency	Is it fast? Low memory, disk, I/O, …?
maintainability	Is it easy to change?
portability	Does it work well everywhere?

(adapted from ISO 9126-1)

What About … Not So Right?

Failure	Defect
An *event*: Something went wrong; unexpected behavior	A *mistake*: Something is likely or certain to cause a failure
Hardware	Cracked solder joint
Network	Flaky router
Data	Data-entry errors
User	Hangover
Software	Bugs!

Manual Testing

Basic Debugging

- You make assumptions about what your code does
- Sometimes, you are wrong
- Check assumptions against reality with **`print()`**

```
my %counts;
while (my $word = <INPUT_FILE>) {

    if (exists $counts{$word}) {
        $counts{$word} = 1;

    } else {
        $counts{$word}++;

    }
}
```

Basic Debugging

- You make assumptions about what your code does
- Sometimes, you are wrong
- Check assumptions against reality with **`print()`**

```
my %counts;
while (my $word = <INPUT_FILE>) {
    print "$word, $counts{$word}\n";
    if (exists $counts{$word}) {
        $counts{$word} = 1;
        print "exist: $word, $counts{$word}\n";
    } else {
        $counts{$word}++;
        print "not: $word, $counts{$word}\n";
    }
}
```

Better Debugging With `print()`

```perl
my $DEBUG = 1; # 0 is no debug, 1 is debug
sub convert {
    my ($from, $to, $value) = @_;
    print "from = $from\n" if $DEBUG;
    print "to = $to\n" if $DEBUG;
    print "value = $value\n" if $DEBUG;

    my $meters = $value * $UNITS{$from};
    print "meters = $meters\n" if $DEBUG;

    my $result = $meters / $UNITS{$to};
    print "result = $result\n" if $DEBUG;
    return $result;
}
```

Debugging Tips

- Use binary search
- Make ***one*** change at a time
- Investigate ***all*** unexpected results
- Understand error messages
- Challenge your assumptions

Debugger

Common Debugger Features

- View code
- Run live code, line-by-line
- Examine variables
- Run and stop at breakpoints
- Stack traces
- Watch points

The Perl Debugger

```
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
Debugger controls:                         L            List break/watch/actions
  o [...]     Set debugger options         t [expr]     Toggle trace [trace expr]
  <[<]|{[{]|>[>] [cmd] Do pre/post-prompt b [ln|event|sub] [cnd] 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         ![!] syscmd 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.
```

Automated Testing

Automated Testing

- Write software to test (other) software
- Humans vs. machines
- Types of automated tests
  - **Unit tests.** Parts of one script
  - **Functional tests.** Whole script (from outside)
  - **Performance tests.** *(maybe later)*

How to Create Test Cases

- **Normal cases** *(just a few)*
  - `c2f(50)` => `122`
  - `valid_number('42')` => *true*
- **Error cases** *(where you expect failure)*
  - Bad arguments: `c2f('abc')`, `c2f()`, `c2f(50, 42)`
  - Range errors: `$country{'ZZZ'}`, `read('zzzzz')`
- **Tricky cases** *(ones that are hard to get right)*
  - `fix_operators('$foo= 6;')` => `'$foo = 6;'`
- **Boundary cases** *(between normal and error/tricky)*
  - `valid_number('123abc')` => *???*

Test::Simple

Use **ok()** to write a test with one boolean expression

```
sub valid_number { … }     # => boolean
sub c2f { … }              # => number

use Test::Simple tests => 6;

ok(valid_number(42), 'num 42');
ok(valid_number(34.5), 'num 34.5');
ok(not valid_number('abc'), 'num abc');

ok(c2f(0) == 32, 'c2f 0');
ok(c2f(-40) == -40, 'c2f -40');
ok(not defined(c2f('x')), 'c2f x');
```

Test::More

```
use Test::More tests => 6;

ok(valid_number(42), 'num 42');
is(c2f(0), 32, 'c2f 0->32');
isnt($num_errors, 0, 'errors occurred');
like($data[0], qr/^\d+$/, 'number out');
unlike($result{$i}, qr/error/i, 'result');
diag("current value of name = $name");

SKIP: {
  skip('no file', 1) unless -e $file;
  ok(read_file($file), 'file ok');
};
```

Testing a Standalone Script

```
use Getopt::Long;
GetOptions('test' => \&run_tests);

#  Write your main script & subroutines here

sub run_tests {
    require Test::More;
    Test::More->import;
    plan(tests => nnn);

    #  Write test cases here (e.g.,  ok( ) …)

    exit 0;
}
```

Unit Testing Tips

- Test logical chunks of code — usually subroutines
- Aim for *reasonable* coverage
- Run often!
  - After every (significant) change
  - Before you use, hand in, commit, …
- Reproduce bugs as tests *before* fixing

Test-First Development

- Radical idea: Write tests **FIRST**

- Then write code until tests pass
- Clarifies and documents design
- And of course… is useful for testing!

http://junit.sourceforge.net/doc/testinfected/testing.htm

Last 2 Slides…

Other Scripting Languages

- “Try it out” and printing/logging always work
- Most have debuggers and/or interactive modes
- Unit testing:
  - Most others are based on jUnit
  - Expect similar and richer assertions
  - Introspection rocks!

Homework

- Write two fun (!) new subroutines
- Write unit tests using **`Test::More`**
- Use pattern from slide to make **`--test`** work
- Code should pass all tests!