2. Regular Expressions – AS.410.698.81.SU16 Bioperl

Basics of Regular Expressions

Regular expressions are a very powerful (and sometimes very confusing) way of searching for **pattern matches** in text. They are one of Perl's strongest and best known features. Mastering regular expressions is essential to mastering Perl.

Basics are very simple: you use the '`=~`' operator to associate a regular expression with a string. This construct returns true if the regular expression matches the string, and false if it doesn't.

The interesting part is in how you specify the regular expression.

The most basic rule: a character in a regexp matches that same character:

```
#! /usr/bin/perl

use strict;
use warnings;

my $string = "foobarbaz";

if( $string =~ m/bar/ ) {
  print "match\n";
}
else { print "no match\n" }
```

```
$ ./regexp1.pl
match
```

Metacharacters

If all **regexps** let you do was specify exact matches, they wouldn't be very exciting. Fortunately, they let you specify much much ***MUCH*** more than that. Unfortunately, the syntax and terminology used gets a bit thick.

In order to be able to specify that something in a regexp isn't supposed to literally match something, you need a way to let Perl know that you're trying to do something fancy. Enter **metacharacters**.

```
\   # meta meta
|   # alternation
()  # grouping
[]  # character classes
.   # wildcard
*   # quantifier
+   # quantifier
?   # quantifier
{}  # quantifiers
^   # anchor at beginning of string
$   # anchor at end of string
```

The effect of the `\` metacharcter depends on what it proceeds:

- Placed in front of another metacharacter, it **disables** the meta effect.
- Placed in front of certain letters, it **enables** a new meta effect.
- Placed in front of any other character, it has no effect.

Alternation

Alternation works like it did with the `case` statement in shell: you specify one or more alternatives, separated with pipes. If any one of the alternates matches, the regular expression matches. In order to specify the beginning and end of the alternation pattern, you use parens:

```
/b(ar|az)/ # matches 'bar' or 'baz'
```

Character Classes

Character classes are like alternation, but for just one character and one position at a time, using square brackets to hold the possible matches:

```
/ba[rz]/ # matches 'bar' or 'baz'; same as /ba(r|z)/
/[fg]oo/ # matches 'foo' or 'goo'
```

```
/[fg]ooba[rz]/ # matches foobar, foobaz, goobar, goobaz
/[a-z]oo/
/[a-zA-Z]oo/
/[^f]oo/
```

Related to the character classes are some of the special metacharcters formed by putting '\' in front of certain letters -- they're like "mega" character classes:

```
\d  == match any digit ( [0-9] )
\D  == match any nondigit ( [^0-9] )
\s  == match any whitespace character (newline, space, tab, etc.)
\S  == match any non-whitespace character
\w  == match any "word" character
\W  == match any "non-word" character
```

Position Matching

The **position operators** let you **anchor** a match to the beginning or end of a string.

```
^   # beginning of string
$   # end of string

/foo/   # matches 'foo' , 'foobar' , 'barfoo'
/^foo/  # matches 'foo' , 'foobar' , NOT 'barfoo'
/foo$/  # matches 'foo' , 'barfoo' , NOT 'foobar'
/^foo$/ # matches 'foo' , NOT 'barfoo', NOT 'foobar'
```

Wildcards

The **wildcard** is like, well, wild cards in card games: it matches anything you want. It's basically a character class that includes all possible characters.

```
.   # wildcard
```

A single '.' matches a single position in the string. To get more sophisticated matches, you must use **quantifiers**.

Quantifiers

If you want to match more than one character at a time, then you need to use **quantifiers**, which specify how many times to match the **preceding character:**

```
*   # quantifier
+   # quantifier
?   # quantifier
{}  # quantifiers

/a*/         == match 'a' zero or more times
/a+/         == match 'a' one or more times
/a?/         == match 'a' zero or one times
/a{MIN,MAX}/ == match 'a' at least MIN and not more than MAX times
/a{MIN,}/    == match 'a' at least MIN times
/a{COUNT}/   == match 'a' exactly COUNT times
```

Note that quantifiers can apply to **character classes** and to **things grouped with parens**:

```
/f[oa]*/      # matches 'f', 'fo' , 'fa' , 'foo' , 'foa', etc
/f[oa]+/      # matches 'fo' , 'fa' , 'foo' , 'foa', etc
/b(ar|az)?/   # matches 'b' , 'bar' , 'baz'
/f(oo){1,3}/  # matches 'foo' , 'foooo' , 'fooooo'
```

Modifiers

You can also supply **modifiers** that affect how the match is calculated:

```
/i == ignore letter case when figuring matches
/g == globally find all matches
/m == let ^ and $ match next to embedded \n
/s == let . match \n
```

There are others, but they will be introduced later, after we've covered a bit more ground. (See *Programming Perl*, chapter five, if you can't wait.)

Capturing Matches

Suppose you want to **capture** a match -- find out exactly what in the string was matched. You can do this with the paren grouping characters.

After a successful match, a special variable is set, containing the characters that were matched by each pair of parentheses. The first one is `$1`, the second is `$2`, and so on. Observe:

```
my $string = "this is an example";
$string =~ /this (\w+) an (\w+)/;
print "$1\n"; # prints 'is'
print "$2\n"; # prints 'example'
```

This technique is frequently used to **parse** data.

Substitution

There are other things you can do with regexps, other than simple matching. One of them is **substitution** -- match a section of a string, and then substitute some other text for the matched portion:

```
my $string = "foobar";
$string =~ s/bar/baz/;
# $string is now "foobaz"

# can use $1, $2 in right-hand side of s/// expression:
$string =~ s/(.{3})(.{3})/$2$1/;
# $string is now 'bazfoo'
```

Modifiers also apply to the `s///` type of matches:

```
/i == ignore letter case when figuring matches
/g == globally find all matches and apply substitution to all
/m == let ^ and $ match next to embedded \n
/s == let . match \n
```

Debugging Regular Expressions

Getting a complicated regular expression to work properly can be frustrating -- especially when you're trying to figure out how to parse a complicated data format.

Generally, the best way to approach this situation is to write a test script, just checking the regexp. Start out with a very simple form of the expression, and make sure you can get it to work. Once you've got that working, add the next piece, and test. Repeat until you have the whole expression, and you're satisfied that the regular expression is working the way you want it to.

One good way of making sure you didn't miss anything is to process your whole input file through the regular expression, just checking that your match always works:

```
open( my $IN , '<' , $file ) or die( "can't open $file ($!)" );
while( <$IN> ) {
  chomp;
  die "failed to match:\n$_\n" unless /REGEXP/;
}
close( $IN );
print "everything matched.\n";
```

Exercises

Open a shell session to the server and write some simple scripts that use regular expressions. A good file to test regular expressions on is `/usr/share/dict/words`, which is used by spellchecking programs. It's about 230,000 words, one word per line. (Is this a good file to **slurp** or not?)

How would you write a script that prints out every line in a file that matches a particular regular expression? How would you write a script to print out every line that *doesn't* match a regexp?

What happens if you **nest** groups of parens in a regexp? What gets captured by each set of parens, and which variable does it end up in?

Table of Contents

What's A Function?

A **function** (also called a **subroutine**) is a named, independent chunk of code that performs one or more tasks. It may accept **arguments** (also called **parameters**) to modify how it behaves, and it may or may not have a **return value**.

Syntax

Perl has many **built-in** functions, but it also allows you to define your own -- those are called **user-defined** functions. Here's the generic syntax for doing so:

```
sub FUNCTION_NAME {
  BLOCK
}
```

Valid function names have the same characteristics as valid variable names: start with a character, include only alphanumerics and underscores. (User-defined functions and variables actually aren't that different, when you get down into the guts of Perl.)

Calling Functions

In order to **call** a function, you just provide the function name, followed by a pair of parentheses:

```
#! /usr/bin/perl -w

use strict;
use warnings;

print_something();

sub print_something {
  print "something\n";
}
```

```
$ ./function.pl
something
```

Note that you don't have to *declare* the function before using it. Because of this, most people put their functions at the end of their programs.

That gets into one of the primary reasons to use functions: they keep your code clean, making it easy to see what is going on. (The other primary reason is code re-use -- a/k/a, "don't repeat yourself").

Getting Input To Functions

How do you get inputs into a function? You **pass arguments** by putting them into the parentheses after the function name. The function retrieves them from the special **@_** variable. (Note the similarity to the magic **$_** variable.) Here's an example:

```
#! /usr/bin/perl

use strict;
use warnings;

add( 1 , 2 );

sub add {
  my( $x , $y ) = ( @_ );
  my $result = $x + $y;
  print( "$x + $y = $result\n" );
}
```

```
$ ./args.pl
1 + 2 = 3
```

Named Parameters

Other programming languages support the idea of **named parameters** for functions. Once you start using functions that have lots of parameters, having names for them is nice, because then you don't have to remember the exact order things have to be placed in. Perl doesn't directly support named parameters, but by using a little trick in your functions, you can have them:

```
#! /usr/bin/perl
```

```
use strict;
use warnings;

named_add( y => 1 ,
           x => 2 );

sub named_add {
  my( %args ) = ( @_ );
  my $result = $args{x} + $args{y};
  print( "$args{x} + $args{y} = $result\n" );
}
```

```
$ ./named_args.pl
2 + 1 = 3
```

What happens if you forget to pass in the right thing?

```
sub named_add {
  my( %args ) = ( @_ );
  die "bad args" unless $args{x} and $args{y};
  my $result = $args{x} + $args{y};
  print( "$args{x} + $args{y} = $result\n" );
}
```

Getting Output From Functions

Suppose you want to do something with a function other than `print` something out -- how do you get a function to give you back something useful? With `return`.

```
#! /usr/bin/perl

use strict;
use warnings;

my $result = returning_add( y => 1 ,
```

```
                            x => 2 );
print $result;

sub returning_add {
  my( %args ) = ( @_ );
  my $result = $args{x} + $args{y};
  return( "$args{x} + $args{y} = $result\n" );
}
```

```
$ ./return_add.pl
2 + 1 = 3
```

ALL functions have a return value. If you don't explicitly return something, Perl returns the value of the last expression in the function.

Putting It All Together

Here's a more realistic example. This is the world's simplest Yahteze game:

```
#! /usr/bin/perl

use strict;
use warnings;

my $dice = 6;

print "You rolled: ";
foreach ( 1 .. $dice ) {
  my $roll = int( rand( 6 )) + 1;
  print "$roll ";
}
print "\n";
```

Rewritten using a function for the dice roll part, it might look something like this:

```
#! /usr/bin/perl
```

```
use strict;
use warnings;

my $dice = 6;

print "You rolled: ";
foreach ( 1 .. $dice ) {
  my $roll = roll_dice();
  print "$roll ";
}
print "\n";

sub roll_dice {
  return int( rand( 6 )) +1;
}
```

Exercises

Return to your shell session and play around with functions. See if you can combine functions with regular expressions.

Review some of the Perl code you wrote for the earlier practice sessions. Identify things that could become functions.

Table of Contents

<

Page 3 of 4

>

Instructions

Homework and exams will be submitted electronically. You should create a git repository in your home directory in the `proj` directory, as was demonstrated in class, and then commit your answers to the following questions into the repository. I will check out your work from your repository at the beginning of next week's class and grade that work. You should place the answers to this week's homework questions in a sub-directory called 'week03'. The answer to each question should go into a distinct file or directory, named for the question: `q1`, `q2`, etc. Issues of layout beyond that are up to you, except as specified in the question. In

other words, your answer to question one could be found in `~/proj/week03/q1.txt` or in `~/proj/week03/q1/answer.txt` -- you decide.

Please note: homework will be gathered immediately prior to the start of next class; this will be done programmatically. Deviation from the instructions above may cause your homework to not be properly collected.

Problems

1. Write a script that uses **nested loop constructs** to print the multiplication tables up to 12. Sample output:

```
1   2   3   4   5   6   7   8   9  10  11  12
2   4   6   8  10  12  14  16  18  20  22  24
3   6   9  12  15  18  21  24  27  30  33  36
4   8  12  16  20  24  28  32  36  40  44  48
5  10  15  20  25  30  35  40  45  50  55  60
6  12  18  24  30  36  42  48  54  60  66  72
7  14  21  28  35  42  49  56  63  70  77  84
8  16  24  32  40  48  56  64  72  80  88  96
9  18  27  36  45  54  63  72  81  90  99 108
10 20  30  40  50  60  70  80  90 100 110 120
11 22  33  44  55  66  77  88  99 110 121 132
12 24  36  48  60  72  84  96 108 120 132 144
```

   (Don't worry if you can't get things to line up neatly like that; if you're deeply bothered by nonalignment, `perldoc -f printf` and `perldoc -f sprintf`.)

   (5 pts)
2. Write the script from the previous question as a function that takes an argument for the size of the matrix. So that:

```
print mult_table( 5 );
```

   would produce

```
1   2   3   4   5
2   4   6   8  10
3   6   9  12  15
4   8  12  16  20
5  10  15  20  25
```

(Again, don't worry about getting things to line up correctly.)

(5 pts)

3. Write a script to generate random DNA sequences of a given length.

```
$ ./random.pl
AAACGCAAGGGTATGTATAACAAGGCTTGCTCGATGTTGGAAGGGTGGAC
```

(5 pts)

4. Rewrite the script from the last question as a function, taking an argument for the length of the sequence that is returned. Write the function such that a second, optional argument, if provided, will cause the function to produce a sequence of a random length between 1 and the first argument.

(5 pts)

5. Use the program from the previous problem to generate 10 random DNA sequences of at least 50 bases. Save them in 10 distinct files (don't forget to remove any header you added). Write a program that prints the names of the files that contain sequences that have at least one run of four identical bases in them.

```
$ ./base_find.pl
T run found in ./seq2
G run found in ./seq4
T run found in ./seq6
G run found in ./seq7
T run found in ./seq8
```

(5 pts)

6. Write a regexp debugger: Code which asks for a regular expression, then tests inputs against that regular expression and reports whether or not they match. It should also support an 'exit' command to allow quitting. Sample output:

```
$ ./regexp_debug.pl
Enter regexp: oo
Enter string or 'exit' to exit; foo
Match!
Enter string or 'exit' to exit; bar
No match!
Enter string or 'exit' to exit; exit
bye bye!
```

(10 pts)