Regular Expressions

The Purpose

- Regular expressions are the main way Perl matches patterns within strings. For example, finding pieces of text within a larger document, or finding a restriction site within a larger sequence.
- There are 3 main operators that use regular expressions:

  1. matching (which returns TRUE if a match is found and FALSE if no match is found.

  2. substitution, which substitutes one pattern of characters for another within a string

  3. split, which separates a string into a series of substrings
- Regular expressions are composed of characters, character classes, metacharacters, quantifiers, and assertions.

Match Operator

- If you want to determine whether a string matches a particular pattern, the basic syntax is:

```
$str = “This is a string”;
if ($str =~ /ing/ ) { print “match”;  }
else { print “no match”; }
```

- Two important things here: “=~” is the match operator. It means “search the preceding string ($str) for the pattern between the slashes (“ing” here).
- Simple matching returns TRUE or FALSE. Anything you put between the slashes will be used as a test pattern, which Perl attempts to match with the string.

Literal Matching

- Note that the pattern must be an EXACT match. The following won't work because there is a space between the "n" and the "g".

```
$str = "This is a string";
if ($str =~ /in g/ ) { print "match";  }
```

- You can put a variable in place of the match string:

```
$pattern = "ing";
if ($str =~ /$pattern/ ) { print "match";  }  # works!
```

- One simple modification: an "i" after the second slash makes the matching case-insensitive. The default is case-sensitive.

  For example:

```
$str = "This is s string";
$str =~ /this/ ;   # does NOT match
$str =~ /this/i ;  # matches
```

Basic Quantifiers

- Quantifiers are placed after the character you want to match.
- * means 0 or more of the preceding character
- + means 1 or more
- ? Means 0 or 1
- For example:

```
my $str = “AACCGG”;
$str =~ /A+/;  # matches AA
$str =~ /T+/;  # no match
$str =~ /T*/;  # matches 0 or more T’s
$str =~ /Q*/;  # matches: 0 or more Q’s
```

- Matching positive or negative 23:

```
$str =~ /-?23/;  # i.e. 0 or 1 –’s
```

More Quantifiers

- You can specify an exact number of repeats of a character to match using curly braces:

```
$str = “doggggy”;
 $str =~ /dog{4}y/;  # matches 4 g’s
 $str =~ /dog{3}y/;  # no match
 $str =~ /dog{3}/; # matches--note no trailing “y” in the
```
pattern.

- You can also specify a range by separating the minimum and maximum by a comma within the curly braces:

```
$str =~ /dog{1,5}y/; # matches 1,2, 3, 4, or 5 g’s
```

- You can also specify a minimum number of characters to match by putting a comma after the minimum number:

```
$str =~ /dog{3,}; # matches 3 or more g’s
```

Grouping with Parentheses

- If you want to match a certain number of repeats of a group of characters, you can group the characters within parentheses. For example, /(cat){3}/ matches 3 reps of “cat” in a row: “catcatcat”. However, /cat{3}/ matches “ca” followed by 3 t’s: “cattt”.
- Parentheses also invoke the pattern matching memory, to be discussed later.

Basic Metacharacters

- Some characters mean something other than the literal character.
- For example, “+” means “1 or more of the preceding character. What if you want to match a literal plus sign? To do this, escape the + by putting a backslash in front of it: \+ will match a + sign, but nothing else.
- To match a literal backslash, use 2 of them: \\.

- Another important metacharacter: “.” matches any character. Thus, /ATG…UGA/ would match ATG followed by 3 additional characters of any type, followed by UGA.
- Note that /.*/ matches any string, even the empty string. It means: 0 or more of any character”.
- To match a literal period, escape it: \.
- The “.” doesn’t match a newline. There’s a way around this we will discuss later.
- List of characters that need to be escaped: \ | / ( ) [ ] { } ^ $ * + ? .

Basic Assertions

- An assertion is a statement about the position of the match pattern within a string.
- The most common assertions are “^”, which signifies the beginning of a string, and “$”, which signifies the end of the string.
- For example:

```
my $str = “The dog”;
$str =~ /dog/;  # matches
$str =~ /^dog/; # doesn’t work: “d” must be the first character
$str =~ /dog$/; # works: “g” is the last character
```

- Another common assertion: “\b” signifies the beginning or end of a word. For example:

```
$str = “There is a dog”;
$str =~ /The/ ;  # matches
$str =~ /The\b/ ; # doesn’t match because the “e” isn’t at the end of the
word
```

Character Classes

- A character class is a way of matching 1 character in the string being searched to any of a number of characters in the search pattern.
- Character classes are defined using square brackets. Thus. [135] matches any of 1, 3, or 5.
- A range of characters (based on ASCII order) can be used in a character class: [0-7] matches any digit between 0 and 7, and [a-z] matches and small (but not capital) letter.

  --note that the hyphen is being used as a metacharacter here. You can't use the backslash escape in character classes. To match a literal hyphen in a character class, it needs to be the first character in the class. Thus. [-135] matches any of -, 1, 3, or 5. [-0-9] matches any digit or the hyphen.

More Character Classes

- To negate a character class, that is, to match any character EXCEPT what is in the class, use the caret ^ as the first symbol in the class. [^0-9] matches any character that isn't a digit. [^-0-9] ,matches any character that isn't a hyphen or a digit.

  --here the caret is a metacharacter. to match a literal caret, put it in any position except the first position.
- The only other character inside a character class that must be escaped is the closing square bracket. To match it as a literal, escape it with a backslash:[0-9\]] matches any digit or].
- Quantifiers can be used with character classes. [135]+ matches 1 or more of 1, 3, or 5 (in any combination). [246]{8} matches 8 of 2, 4, and 6 in any combination.

Pre-defined Character Classes

- Several groups of characters are so widely used that they are given special designators. These do not need to be put inside square brackets unless you want to include other characters in the class.
- \d = any digit = [0-9]
- \s = whitespace (spaces, tabs, newlines) = [\t\n]
- \w - word character = [a-zA-Z0-9_]
- The negation of these classes use the capital letters: \D = any non-digit, \S = any non-whitespace character, and \W = any non-word character.

Alternatives

- Alternative match patterns are separated by the “|” character. Thus:

  $str = “My pet is a dog.”;

  $str =~ /dog|cat|bird/; # matches “dog” or “cat” or “bird”.
- Note there is no need to group the characters with parentheses. Use of the | implies all of the characters between the delimiters.

Memory

- It is possible to save part or all of the string that matches the pattern. To do this, simply group the characters to be saved in parentheses. The matching string is saved in scalar variables starting with $1.

  ```
  $str = "The z number is z576890";
  $str =~ /is z(\d+)/;
  print $1;  # prints "567890"
  ```

- Different variables are counted from left to right by the position of the opening parenthesis:

  ```
   /(the ((cat) (runs)))/ ;
  ```

  captures: $1 = the cat runs; $2 = cat runs; $3 = cat; $4 = runs.

- The numbered variables only work within the smallest block (delimited by {}) that contains the regular expression. They are reset to undef outside that block.

More on Memory

- if you are still within the regex, the memory variables are called \1, \2, etc. instead of $1, $2, etc. These variables work within a match to allow duplication of a pattern. For example, /([AG]{3})CCGG\1/ matches AGGCCGGAGG. Note that the two sides, the pattern in the parentheses and the duplication in \1, must match exactly, even though the pattern in parentheses ia a character class.

- If several alternatives are captured by parentheses, only the one that actually matches will be captured:

```
$str = "My pet is a cat";
$str =~ /\b(cat|dog)\b/;
 print $1; will print "cat".
```

Greedy vs. Lazy Matching

- The regular expression engine does “greedy” matching by default. This means that it attempts to match the maximum possible number of characters, if given a choice. For example:

  ```
  $str = “The dogggg”;
  $str =~ /The (dog+)/;
  print $1;
  ```

  This prints “dogggg” because “g+”, one or more g’s, is interpreted to mean the maximum possible number of g’s.
- Greedy matching can cause problems with the use of quantifiers. Imagine that you have a long DNA sequence and you try to match /ATG(.*)TAG/. The “.*” matches 0 or more of any character. Greedy matching causes this to take the entire sequence between the first ATG and the last TAG. This could be a very long matched sequence.
- Lazy matching matches the minimal number of characters. It is turned on by putting a question mark “?” after a quantifier. Using the examples above,

  ```
  $str =~ /The (dog+?)/; print $1;  # prints “dog”
  ```

  and /ATG(.*?)TAG/ captures everything between the first ATG and the first TAG that follows. This is usually what you want to do with large sequences.

A Few Examples

- 1. Finding blank lines. They might have a space or tab on them. so use /^\s*$/

- 2. matching letters only. The problem is, \w is digits and underscore in addition ot letters. You can use /^A-Za-z$/, which requires that every character in the entire string be a letter: Or try /^[^\W\d_]$/. For DNA bases, /^ACGTacgt$/ works, but realize that “N” is often used as a wildcard character in sequences.

More

- 3. Words in general sometimes have apostrophe(') and hyphen (-) in them, such as o'clock and cat's and pre-evaluation.. Also, there are some common numerical/letter mixed expressions: 1st for first, for example. So, \w by itself won’t match everything that we consider a word in common English.

- 4. Words can be found within words: "there goes the cat" =~ m/the/ matches the first word, not the fourth, but =~ m/\bthe\b/ matches the word "the" only.

More Examples

- 5. time of day: For example. 11:30.   [01][0-9]:[0-5][0-9] won't work well, because it would allow such impossible times as 19:00 and 00:30. A more complicated construction works better: (1[012] | [1-9]) :[0-5][0-9]. That is, a 1 followed by 0, 1, or 2, OR any digit 1-9.

- 6. Things within a delimiter: e.g. <a href=http://www.bios.niu.edu> You want what's within the tags, everything between <a and >, so you try m/<a(.*?)>/. Note the need for lazy evaluation here, as there is probably more than one tag on the page. But still, this expression picks up the ending >, because it matches .*. So, use m/<a[^>]*>/ That is, zero or more characters that are not >. Usually you don’t really want .*

Still More Examples

- 7. BLAST scores (e-values) can be either decimals or exponents: 0.05 and 2e-40 and both typical values. To capture these numbers, use ([-e.\d]+). Also, sometimes BLAST scores start with e: e-35, for example. Perl doesn't recognize this as a number, so you have to add a leading "1":

```
$score = "1" . $score if substr($score, 0, 1) eq "e";
```

8. Regular expressions don't work very well with nested delimiters or other tree-like data structures, such as are found in an HTML table or an XML document. We will discuss alternatives later in the course.

Alternative Delimiters

- We have been enclosing all of the match patterns between slashes “/”. This is the default method.
- However, you can use almost any pair of non-word characters as delimiters if you precede them with an “m”. Thus all of these work:

  m<cat>, m@cat@ , m\cat\.
- Use of alternative delimiters leads quickly to obfuscation, so I avoid them. Especially, ? and ‘ as delimiters have special meanings.

Split

- So far we have only looked at simple matching, sometimes with the use of memory for capturing parts of the match. Simple matches just return TRUE or FALSE. We will now look at 2 other uses of regular expressions.
- The “split” operator uses regular expressions to determine where to split the string. Every matching sequence is removed and the remaining parts are put into an array:

```
$str = “I have 2 cats and 3 cars at home”;
@arr = split /a[rt]/, $str;
```

@arr has 3 elements: $arr[0] = “I have 2 c” , $arr[1] = “s and 3 c” , and $arr[2] = “s at home”.

Substitution

- A common problem is the need to substitute one piece of text for another, or to remove certain parts of text. This is done with the substitution operator s///.
- The basic syntax:

  ```
  $str =~ s/initial_pattern/substituted_chars/;
  ```
- Note the initial “s”, followed by a regular expression (the first argument), followed by the group of characters that are to be substituted in (second argument).
- For example:

  ```
  $str = “A cat ia a nice pet”;
  $str =~ s/cat/dog/;
  print $str;  # prints “A dog is a nice pet”
  ```

More Substitution

- The first argument in a substitution is a regular expression, using the same metacharacters, assertions, alternatives, character classes and parentheses as pattern matching. Capturing groups of characters within parentheses also works.
- The second argument, the pattern that is substituted in, is NOT a regular expression. rather it is basically a double quoted string without the quotes. You can do variable substitution, and especially you can use $1, $2, etc. captured form the first argument.   But, only 1 string is going to be substituted in. Things like character classes and alternatives make no sense for the second argument.

Still More Substitution

- By default, substitution only makes one change in a string. If you want to change all examples of the matching pattern, you need to add a “g” on the end:

```
$str = “A cat is a cat is a cat”;
$str =~ s/cat/dog/;  # gives “A dog is a cat is a cat”
$str =~ s/cat/dog/g; # gives “A dog is a dog is a dog”
```

- As for simple matching, substitution can be made case-insensitive by adding an “i” to the end:

```
$str = “Cat”;
$str =~ s/cat/dog/;   # no changes made; “cat” doesn’t match “Cat”
$str =~ s/cat/dog/i;  # $str is now “dog”
```

- You can also use substitution to remove characters. Thus s/[^ACGT]//g finds any character that isn’t A, C, G, or T and replaces it with nothing.

Substitution and Assignment

- Note that unlike many Perl operators, the substitution operator directly works on the string being matched. Thus, $str =~ s/cat/dog/ results in $str being changed.
- If you want to keep the original string and also produce an altered string, you need to do an assignment as well as a substitution. A problem is that =~ has a higher precedence than =, so a construction like this is used:

```
($newstr = $oldstr) =~ s/cat/dog/;
```

- First, $oldstr is copied into $newstr, then $newstr is substituted. $oldstr is not changed by this at all.
- If you change the parentheses:

```
$newstr = ($oldstr =~ s/tcat/dog/ );
```

or don't use them at all, substitution is done on $oldstr, and $newstr gets the number of substitutions done (i.e., s/// is being used in a scalar context).

Transliteration

- The transliteration operator tr/// looks a lot like the substitution operator, but its function is quite different. Transliteration converts EACH instance of an individual character in the first argument to the character in the same position in the second argument. For example:

  ```
  $str = "ACCGTTAC";
  $str =~ tr/ACGT/TGCA/;
  ```

  $str is now TGGCAATG, which is the complement of this DNA sequence. (You can also use "$str = reverse $str;" to get the reverse-complement).
- Another useful transliteration: converting to all uppercase:

  ```
  $str =~ tr/a-z/A-Z/;
  ```
- The first and second arguments are both just lists of individual characters to be replaced. The sequence in which they appear within the arguments is irrelevant; there are no character classes or alternatives with tr///. Also, variable interpolation doesn't work.

Counting Characters with Transliteration

- If the second argument is empty, the characters in the first argument are simply replicated; no actual changes occur in the string. However, tr/// returns the number of characters matched, so this is a good way to count the number of occurrences of given characters.

```
 $str = “AGCCTNNNCGTTANTA”;
$num = ($str=~ tr/ACGT// );
```

  # returns the number of A, C, G and T, without counting the N’s.

- Note the position of the parentheses, which is different from a similar usage with the s/// operator. The parentheses could be left off here, because =~ has a higher precedence than =.

Progressive Matching and Position

- Sometimes you want to find all instances of a pattern and give their positions. This is done with a while loop and a “g” at the end of the match pattern (g for “global”).

  ```
  while ($str =~ /ATG/g ) { }
  ```

- The position of the next character after the end of the matched string is found using “pos” followed by the name of the string variable being matched:

  ```
  while ($str =~ /ATG/g ) {
      my $position = pos $str;
      my $start_position = $position - 4;
      print “$start_position, “;
  }
  ```

- In this construction, the regular expression engine starts at the beginning of the string and moves down it looking for matches. At each match, the body of the while loop executes, and the position of the match is reported with “pos”. Then, the regex engine starts up again, just after the position of the previous match and looks for the next match. This continues until the string is exhausted.

And More!

- There are plenty more regular expression tricks and operations, which can be found in *Programming Perl*, or, for the truly devoted, *Mastering Regular Expressions*.