Perl Programming for Biologists

D. Curtis Jamison
Center for Biomedical Genomics and Informatics
George Mason University
Manassas, Virginia

WILEY-LISS

A JOHN WILEY & SONS, INC., PUBLICATION

Chapter 6
String Manipulation

The most useful aspect of Perl is the string manipulation capabilities built into the language. Perl provides both simple array-based character manipulation commands and mind-bogglingly powerful regular expression pattern matching and string manipulation commands. Together, these commands allow the programmer to search and transform strings with incredible ease, and is one of the primary reasons to choose Perl as a programming language.

6.1 Array-Based Character Manipulation

As their name implies, the array-based character manipulation commands treat a string as if it were an array of single characters. This is the model used by many other languages such as C and Fortran. An array-based view of a DNA sequence would look like Figure 6.1. Each slot of the array contains a single letter of the sequence, and the sequence can be manipulated using the indices associated with each letter.

Of course, strings are scalars, and the indices associated with each letter are an artificial external construct based the value of the string rather than an inherent characteristic of the data structure. Therefore, we need to use specialized commands to utilize the imaginary indices.

To get the index associated with a substring, we use the eponymous index() command:

```
index(STRING, SUBSTRING, POSITION)
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
a	c	t	g	g	t	g	a	t	g	c	c	t	t	a	c	g	t	a	t	g	c	c

Figure 6.1 An array of characters

which takes the STRING and searches for the SUBSTRING starting from the specified POSITION. A good example of how to use the index command is the process of searching a long sequence for a start codon:

```
$seqStart = index($sequence, 'ATG');
```

In this example, we are looking through the string contained in $sequence for a string of letters corresponding to 'ATG'. The default value for the starting position is the beginning of the string, so we can safely leave it out. The number that ends up in $seqStart is the index of the character at the start of the match.

We can use the index and the POSITION variable to look through a genomic sequence for all start codons:

```
$position = 0;
while (($position = index($sequence, 'ATG', $position)) >= 0) {
  push(@seqStart, $position);
  $position++;
}
```

The expression controlling the while loop is true when there is an ATG in the sequence downstream of $position: index() returns −1 otherwise. The control expression also modifies $position to be the next index of ATG, which we push onto an array of start positions. Finally, we push our search start position ahead by one so we don't keeping finding the same ATG over and over again. Note that we start $position at 0: remember that strings are like arrays in that their numbering system begins at 0.

The index() command searches left to right. To search right to left, the completely analogous rindex is used to look for start codons in the reverse strand:

```
$position = length($sequence);
while (($position = rindex($sequence, 'CAT', $position)) >= 0) {
  push(@seqStart, $position);
  $position--;
}
```

In this code, there are three changes. First, we are using rindex() to search from right to left so we autodecrement the position. We also reverse complemented our search string, changing it from 'ATG' to 'CAT'. And finally, we used the length() command to give us the starting position (the far right end).

Often we want to extract a portion of a string. We can create a new string by excising a chunk from an existing string using the substr() command:

```
substr(STRING, START, LENGTH)
```

The substr() command returns a new string excised from STRING, beginning at START and running for LENGTH characters. If START is negative, substr begins counting from the end of STRING. So, if we wanted to store the first 60 nucleotides following the start codons in our genomic sequence, we could write

```
$position = 0;
while (($position = index($sequence, 'ATG', $position)) >= 0) {
  push(@seqs, substr($sequence, $position, 60));
  $position++;
}
```

If the sum of the START and LENGTH arguments is greater then the length of the original, the new string will run from START to the end of the original sequence. Similarly, if the LENGTH argument is omitted, the default length is from the position specified by the START argument to the end of the original sequence.

The substr() command is relatively versatile. It can also be used to splice characters into a string by placing it on the left side of the assignment:

```
$string = "my short string";
substr($string, 3, 0) = "not so ";
print $string, "\n";
```

results in

```
my not so short string
```

The string on the right has been inserted into $string, which grows in response. The substr() command has started at position 3 and replaced the substring of length 0 with the string on the right. In addition to inserting, we can actually replace portions:

```
substr($string, 3, 5) = "long and convoluted example";
```

replaces the word `"short"` in the original string and results in

```
my long and convoluted example string
```

The substr() command can be used to append or prepend strings by using a start index of 0 or length(), respectively, and then using a 0 for the length argument. A simpler and faster way to do so is to use the concatenation operator, which is the . symbol. This operator takes the strings on either side and creates a single string by appending the string on the right to the string on the left:

```
$newstring = "new" . "string";
```

We have already seen this operator in action with the concatenation assignment operator:

```
    $retval .= $base;
```

which is the short-hand equivalent of

```
    $retval = $retval . $base;
```

There are two major caveats to remember when working with array-based character operators. First, there is a special Perl variable defined that can alter the numbering system of the character indices. Certain misguided programmers, in an attempt to make Perl act more like inferior programming languages, will change the value of the starting index so that the Perl commands work more like the ones they are used to. In general, this qualifies as a very bad idea, and if you come across code that has done this, it is best to throw it away and start all over again because you can't be sure of what other atrocities the programmer has committed.

The second caveat is much more likely to rear up and bite unsuspecting novice programmers. Perl (and computers in general) treat all characters as equal, whether or not the character is seen when it is sent to the printer. Thus, a newline character `"\n"` counts as a character when querying the length of a string. So does a space or a tab character.

Typically, one runs into newlines when reading text from a file. Because this is a very common problem, Perl has the chomp() command:

```
    chomp ($VAR)
    chomp (LIST)
    chomp
```

The chomp command removes the character at the end of the string if it matches the character stored in the special variable $/ (known as the record input separator, it is set to `"\n"` by default). If chomp is given a LIST, it removes the trailing $/ character string from each string in the list. The default variable that chomp works on is the $_ variable, which we will see in great detail next. A multipurpose variable, $_ is the default target for all text operations, including character-based methods, regular expression searches, and file input.

6.2 Regular Expressions

As powerful as the array-based character manipulation commands are, they pale in comparison to the regular expression operators. These operators perform pattern matching using regular expressions, and can be used to search, substitute, and transform strings of any length. Regular expressions can be daunting, because they often look like they were composed by a random short-circuit from the keyboard, but once you get the hang of them they become almost second nature, and they serve as one of the key foundations of Perl programming.

There are only four regular expression operators. They are

```
[m]/PATTERN/[g][i][o]
s/PATTERN/PATTERN]/[g][i][e][o]
tr/PATTERNLIST/PATTERNLIST/[c][d][s]
y/PATTERNLIST/PATTERNLIST/[c][d][s]
```

These operators look a little odd, because they were inherited from other programming languages, and have retained their old-fashioned appearance, but they work exactly like any of the comparison operators and functions: they take arguments, perform a task, and return a value.

The m operator is the match operator. It looks for the PATTERN in a string. The s operator is the substitute operator, and it finds the first PATTERN and replaces it with the second PATTERN. Finally, the tr and y operators are synonymous, taking a PATTERN from the first PATTERNLIST and replacing it with the corresponding PATTERN from the second PATTERNLIST.

Both the return value and the behavior of each operator can be modified by one or more optional switches:

```
[c]omplement
[d]elete
[e]valuate
[g]lobal
[i]nsensitive to case
[o]nly evaluate once
[s]queeze multiple characters
```

In practice, the g and i switches are typically the most useful, and the others are very rarely seen. And, of course, the return value is modified by whether you are using the regular expression operator in a scalar or an array context.

By default, the regular expression operators use the $_ variable to get the string value to work with. Because $_ is the default variable for reading from files and any other operation that produces a string value, the default is typically quite useful. For example:

```
if (m/ATG/) {print "Start codon found\n";}
```

looks at the string in $_ and returns true or false. However, the string we want to search in often isn't in the $_ variable. Rather than assigning the string into the $_ variable, we can make the match operators use a different variable as the target string by using the =~ operator:

```
$mySeq = RandomSeq(250);
if ($mySeq =~ m/ATG/) {print "Start codon found\n";}
```

In practice, your code will almost always be clearer to understand if you assign the target string into a variable and use the =~ operator.

6.2.1 Match

The match operator will return a 1 if it is used in a scalar context (such as a loop conditional). Thus, in the example above, when we looked through the string in

$mySeq, the match operator returned either a true or a false value depending on if it found any start codons.

On the other hand, we might want to know how many start codons are in the sequence. To do this, we can take advantage of the fact that when the match operator is used in an array context with the [g]lobal option, it will return an array of the pattern matches:

```
$mySeq = RandomSeq(250);
@starts = $mySeq =~ m/ATG/g;
print "sequence has" . $#starts . "start codons\n";
```

While the second line looks rather strange, it really is not very difficult to understand. The trick is that the =~ operator and the match operator have a higher precedence than the assignment operator. Thus, the expression on the left side of the assignment operator is evaluated first, with the [g]lobal option causing the match operator to find all the occurrences of "ATG" in $mySeq, and then creating a list out of them because of the array context. Then we simply use the $# notation to count how many items are in the @starts array.

The regular expression operators will only produce a list if the [g]lobal switch is used in an array context. If the operator is used in a scalar context, the operators will only return 1 or the undef value. Similarly, even if the operator is used in an array context, it will only return 1 or the undef value if the [g]lobal option has been omitted. In the latter case, we get an array of length one that contains either a 1 or the undef value as the first value.

The match operator is used so commonly that Perl has given us a shortcut. It turns out the leading m is optional, and any pattern between two forward slashes tells Perl to perform the match. Thus we could have written our start codon matching code as

```
@starts = $mySeq =~ /ATG/g;
```

and gotten the same result. Most Perl programmers will omit the m.

However, there is a downside to the shortcut. If there are forward slashes in the pattern you are trying to match (like a Unix directory path, for example), they will interfere with the regular expression. To get around this, you can use any pair of nonalphanumeric characters as the pattern delimiters along with the m:

```
m#/usr/local/bin#
```

will match any line containing the Unix path to a particular directory.

If the pattern contains a pattern that looks like it might contain a variable, Perl attempts to interpolate the variable to create the pattern. So we can make our matching pattern easier to change (at the cost of some clarity and performance) by setting a variable to contain the search string:

```
$mySeq = RandomSeq(250);
$myStart = "ATG";
push (@starts, $mySeq =~ /$myStart/g);
print "sequence has" . $#starts . "start codons\n";
```

In addition to the [g]lobal switch, the match operator also accepts the [i]nsensitive, which makes all matches in a case-insensitive manner, and the [o]nce switch, which causes any variable interpolation within the pattern to occur only once, which can save much time in loops.

6.2.2 Substitute

The substitute operator is useful for performing large-scale search and replacement. For example, we might want to change our DNA sequence to an RNA sequence:

```
$mySeq = RandomSeq(250);
$mySeq =~ s/T/U/g;
```

This code tells Perl to take the string and replace every T with a U. Like the match operator, we can specify that the substitution take place [g]lobally. We can also use the i and the o switch with the same meanings as before.

A new switch for the substitute operator is the [e]valuate switch. This tells Perl that the replacement pattern is an expression, and the expression should be evaluated before the replacement. To illustrate, we can make our DNA to RNA transcription code a little more complex:

```
$newCodon = ‘U’;
$mySeq = RandomSeq(250);
$mySeq =~ s/T/$newCodon/eg;
```

Note that when using options, it doesn’t matter in what order they appear.

6.2.3 Translate

Perl has two translation operators: tr and y. The two are exactly equivalent, and the reasons for having both lie back in the prehistory of Perl and the language *sed*, from which many of the regular expression syntax derives. For new Perl programmers the tr operator is easier and more sensible to remember.

The tr operator takes the characters in the first pattern and replaces them with the characters in the second pattern. The first character in the matching pattern list is replaced with the first character in the replacement pattern list, the second with the second, and so forth. This is very different from the substitute operator, which assumes that the entire matching pattern is to be replaced with the entire replacement pattern.

A perfect example of using the translate function is to reverse complement a sequence:

```
$mySeq = RandomSeq(250);
$mySeq =~ tr/ACTG/TGAC/;
$revcomp = reverse($mySeq);
```

Using the reverse function on a string returns a reversed string, analogous to using reverse on an array that contained a single letter in each slot.

Also, note that we did not use the [g]lobal option. The tr operator always acts globally. There are three switches that can be used with the tr operator. The [c]omplement switch inverses the search list, interpreting the list as containing every possible character *except* those listed in the search list. Thus

```
tr/ACTG/ /c;
```

replaces every character that is not an A, C, G, or T with a space, which could be a very useful device for cleaning uncertainty characters out of a sequence. Note that the replacement list is shorter than the search list: in cases like this Perl reuses the last character in the replacement list. Be careful not to confuse the [c]omplement option with the biological meaning of nucleic acid complement.

Another switch is the [s]queeze option, which will compress a run of replacement characters into a single character. Thus, if you have a sequence like

```
ACTGGTAxxxxxxxATAGGxxTGAT
```

the command

```
tr/ACTG/ /cs;
```

would result in the string

```
ACTGGTA ATAGG TGAT
```

The tr operator returns a scalar value stating how many replacements were made. If we had assigned the last tr command to a scalar, the variable would have contained 9 as the number of x's we replaced. We can make use of this to count symbols if we make the replacement list the same as the search list:

```
$hydrophobic = tr/TFY/TFY/;
```

would count the number of hydrophobic residues in a protein sequence without actually altering the sequence. Because this is a common task, as we would expect, Perl provides a shortcut to reduce our typing. Leaving out the replacement list causes the search list to be replicated as the replacement list, so we could have written our residue counter as

```
$hydrophobic = tr/TFY//;
```

with no change in meaning. Which brings us to our last switch, the [d]elete switch, which causes any matches to be removed completely:

```
tr/ACTG//cd;
```

causes all the uncertainty codes in our first example to be removed completely.

6.3 Patterns

Thus far, we've begged off a bit on the question of exactly what a pattern is. A pattern is composed of a set of atoms, quantifiers, and assertions. Atoms are the individual characters that make up the pattern, quantifiers are phrases that control how many atoms are seen, and assertions control where the atoms are found. Together, the three aspects of patterns make for an incredibly rich and powerful language.

6.3.1 Atoms

We've seen some examples of patterns that are simple character strings. The character string represents a series of atoms, which are the basic single character matching substrate. For example, when searching for the start codon, we created a pattern of three atoms: A, T, and G. The regular expression operator then looked for those three atoms in the specified order.

Any character that Perl recognizes can be used as an atom. That includes letters, numbers, and spaces. Each character matches itself once, unless it is quantified. Characters protected with a backslash match that character, with the following exceptions:

- \c followed by a character matches the corresponding control character. For example, \cD matches the Control-D character;
- \x followed by a two-digit hexadecimal number matches the character having that hexadecimal value;
- \0 used as a two- or three-digit octal value matches the character with that octal value;
- \n matches a newline;
- \r matches a return;
- \f matches a form feed;
- \t matches a tab;
- \d, \D, \w, \W, \s, \S match special character classes defined below.

6.3.2 Special Atoms

Certain characters and combinations of characters create special complex atoms. Complex atoms can be combined with quantifiers and assertions just like regular atoms. The only trick is that if you want to match on one of these special characters rather than using it as a complex atom, you have to backslash-escape the character.

A . matches any character except the \n character. So the pattern

```
A.G
```

matches ATG, ACG, AAG, and AGG, as well as any other triplet of letters that starts with A and ends with G. You can restrict the atom by giving a list of characters in square brackets:

```
A[ATCG]G
```

will match only valid nucleotide triplets and ignores any triplets that have nonvalid DNA characters.

A hyphen between two characters in square brackets specifies a range of characters to look for. So

```
[0-9]
```

would match any digit in that range. Similarly,

```
[a-z]
```

matches any lowercase letter, and

```
[a-zA-Z]
```

matches any letter no matter what case it is. A ˆ at the front of the list negates the class:

```
[^a-z]
```

matches any character that is not a lowercase letter.

Some ranges are so useful that Perl defines some backslash-escaped characters that match a specific class

- \d matches any digit, the same as [0–9]
- \D matches any nondigit, the same as [ˆ0–9]
- \w matches any alphanumeric character, the same as [0-9a-zA-Z]
- \W matches any nonalphanumeric character, the same as [ˆ0-9a-zA-Z]
- \s matches any white-space character, the same as [\t\n\r\f]
- \S matches any non-white-space character, the same as [\ˆ \t\n\r\f]

Note that \w is not equivalent to \S, and \W is not equivalent to \s. This is because there are symbol and control characters that won't match the \w or \s atom, but will match the \W and \S atom.

6.3.3 Quantifiers

Atoms can be quantified, meaning that a specific number of the atoms must be present. Quantifiers are placed directly behind the atom they are quantifying. The general syntax is

```
{x,y}
```

where x and y are numbers. The quantifier indicates the atom must be matched at least x times but not more than y times. Thus

```
A{100,200}
```

would match runs of at least 100 and no more than 200 A's (the definition of valid polyadenylations) in a file of mRNA sequences.

Leaving the second argument undefined

```
{x,}
```

means the math must occur at least x times, but there is no upper limit. Providing only the first argument

```
{x}
```

means the match must occur exactly x times. Note the difference between the two is the presence of the comma: with a comma the second argument is undefined, whereas without it the second argument is deleted.

Not surprisingly, Perl has several special case quantifier symbols defined for commonly used quantifiers:

- * matches 0 or more times, same as {0,}
- + matches 1 or more times, same as {1,}
- ? matches 0 or 1 times, same as {0,1}

6.3.4 Assertions

Finally, the regular expression language is rounded out by four assertions that control where the matches are located. The ˆ symbol matches the beginning of the string, the $ symbol matches the end of the string, a \b matches at a word boundary, and a \B matches at a nonword boundary. Note that the ˆ and the $ have other meanings as well, and it is important to check the context of the symbol:

```
$seq =~ /^[AUCG]*A{100, 200}$/
```

is a way to validate that $seq contains a mRNA sequence, whereas

```
$seq =~ /[^AUCG]*A{100, 200}$/
```

makes sure there are no A, U, C, or G characters in the match, which more than likely is not what the programmer intended.

6.3.5 Alternatives

Often we run across situations where multiple patterns are valid. For example, if we wanted to look for start codons in bacteria, we should take into account the fact that some species have alternative start codons. We could look twice (or more) using the Boolean OR operator:

```
if (($seq =~ /AUG/) || ($seq =~ /AGG/)) {...}
```

but that is somewhat unwieldy. Perl allows us to collapse the alternatives into a single regular expression by using the | symbol (half an OR):

```
if ($seq =~ /(AUG)|(AGG)/) {...}
```

Any number of alternatives can be separated by the | symbol. Alternatives are evaluated left to right and the evaluation stops on the first positive match, much like the || command.

The parentheses serve to consolidate individual atoms into a single larger atom. This is useful if we want to more create complex matching schemes. For example, we might want to look for CAG expansion repeats in our DNA sequence:

```
/((CAG)+)/
```

The parentheses also have another beneficial side effect. They can be used to extract the last string that matched the expression inside the parentheses. Each pair of parentheses associated with either a + or a * is assigned a number, going from left to right, and a variable consisting of the $ and the assigned number of the parentheses is created to contain the match. Thus, if we used our expression for real, $1 would contain the first CAG expansion matched in the sequence:

```
my $rna = 'UAAGACGUCAGCAGCAGCAGCAGAAAAGCAGCAGAAA';
if ($rna =~ /((CAG)+)/) {
  print "The first expansion is $1 \n";
}
```

which will produce:

```
The first expansion is CAGCAGCAGCAGCAG
```

The numbered variables aren't the only ones created by the regular expression operator. The $ variable contains the entire matched string, whereas $` and $' return everything before and after the match respectively. These variables are scoped to the block that contains the regular expression, so that they cannot be accessed or are changed back to their original status after the block is finished. These variables are undefined if a match is not found.

Chapter Summary

- Array-based character manipulation treats a character string like an array of characters.
- Exact string matches can be found using the index() and rindex() commands.

- Substrings can be extracted using the substr() command.
- Regular expressions use pattern matching to manipulate strings.
- The m operator finds pattern matches.
- The s operator substitutes a string for a pattern.
- The tr and y operators translate one pattern list into another pattern list.
- The $_ variable is the default target for regular expression operators.
- The =~ operator causes the regular expression operator to act upon the string in the variable on the left.

For More Information

```
perldoc perlfunc
perldoc perlre
```

Exercises

1. Write a subroutine that performs the same function as the chop() command.
2. Compare and contrast Prosite motifs to Perl regular expressions.
3. Write a regular expression that will find GAG repeat expansions.
4. Write a regular expression that represents prokaryotic translation start sites.
5. Give examples of when to use the array-based character manipulation commands versus the regular expression commands.
6. Write a subroutine that duplicates the function of the regular expression s/// operator using array-based character manipulation commands. Take into account wild-cards and identifiers.

Programming Challenges

1. A common use of Perl is to read a tab- or space-delimited file into an array, and then manipulate the results. For example, many microarray programs leave the data in Excel spreadsheets that can be exported to tab-delimited text files:

```
Number      Name  ch1_Ratio   ch1_Percent ch2_Ratio
       ch2_Percent
1      Mouse_actin_beta  1     43.366182   1.305944
       56.633818
2      cytochrome_P450_2b9     1     60.364437   0.656605
       39.635563
```

```
3     thrombomodulin    1    51.152543   0.954937
      48.847457
4     cytokine_inducible_protein     1     50.155169
      0.993812    49.844831
5     Glvr-1_mRNA_complete_cds       1     46.36159
      1.156958    53.63841
6     PDE1A2      1    47.743435   1.094529    52.256565
7     Fas-associated_factor_1 1     46.01072    1.173407
      53.98928
```

Write a program that parses the information out of files with this format and print out the gene names and fold-differences in descending order.

2. Write a program that splices out introns from a sequence.
3. Write a program that translates an RNA sequence to a protein sequence, keeping track of codon usage.